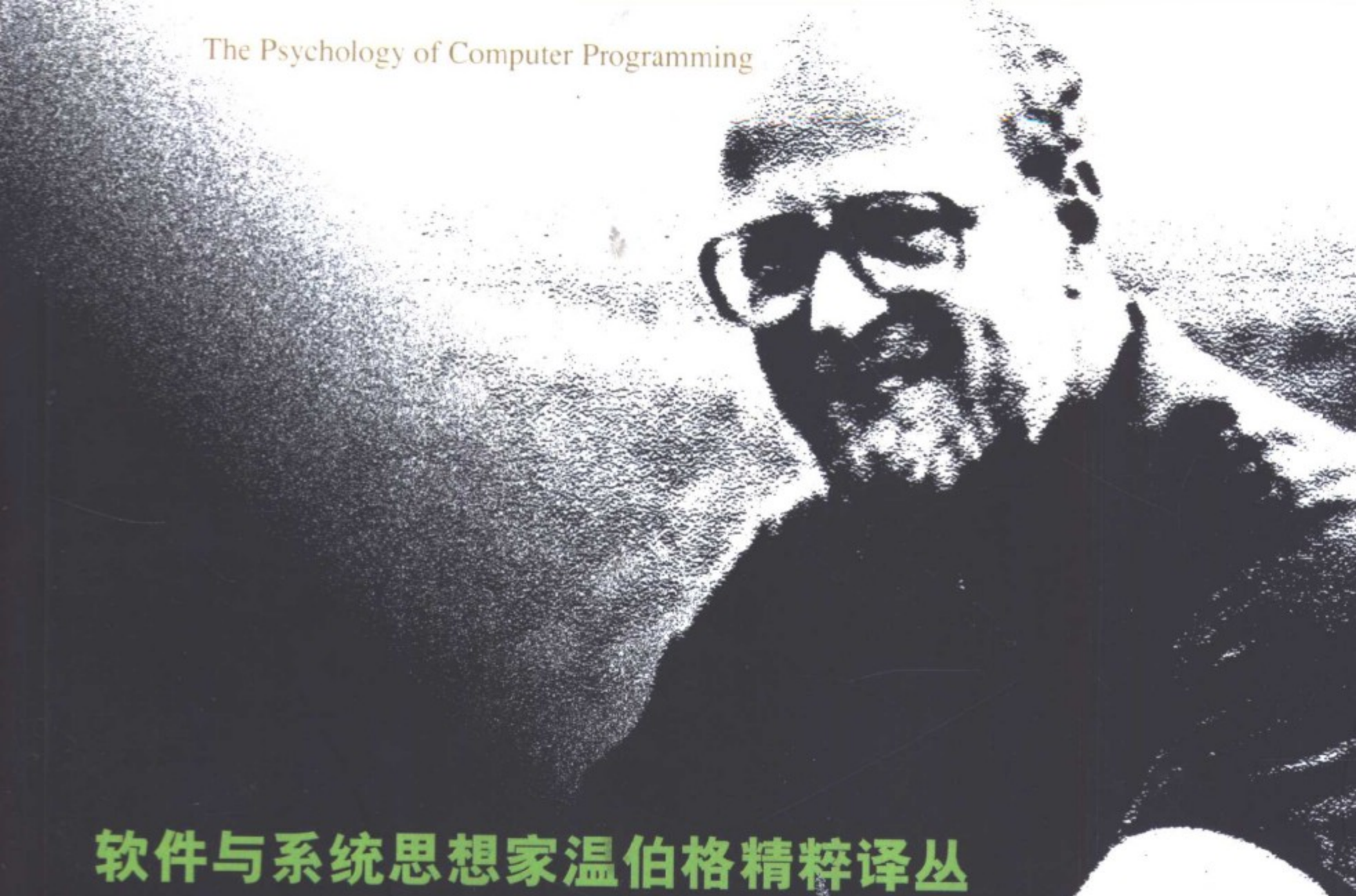
The Psychology of Computer Programming



软件与系统思想家温伯格精粹译丛

程序开发心理学（银年纪念版）

[美] 杰拉尔德·温伯格 /著
邓俊辉 /译

清华大学出版社

清华大学出版社·闻洁编辑室 策划

有史以来计算机编程领域最经典的一部著作。

——Datamation

一本杰作的新生！……不管你是属于20世纪60年代和70年代那一代，还是属于80年代和90年代的新一代，你要感谢你自己去拿起这样一本奇妙的书。一旦你读了它，你应该接着追读温伯格所有其他的书，每本都是一颗宝石。

——Ed Yourdon，Cutter IT Journal

第一本强调计算机编程是个人和团队共同努力过程的图书，并且成为了这个领域的经典图书！……尽管本书写作时采用的是1971年的视角，但是30多年后的今天，这本书还是所有软件开发经理们的必读经典。

——J.J.Hirschfelder，Computing Reviews

本书写成于1969年，1971年出版，自此被不断重印，其重印记录是任何其他计算图书都未能比及的。大多数书中的"关于人的内容"比其中的技术更经得住时间的考验。

——Sue Petersen，
Visual Developer Magazine

温伯格的《程序开发心理学》是我最喜爱的一本关于软件开发的书。……书中有关"无私软件开发"的观点……带动了开放源码思想的崛起。好的软件工程思想要比具体的技术更经久不衰。就这一点而言，没有一本书提供的思想可以与《程序开发心理学》中的思想相比。

——Steve McConnel，IEEE主编，IEEE Software

温伯格的经典之作《程序开发心理学》影响了我的一生。……和《人月神话》一样，《程序开发心理学》同样是一本经久不衰的经典之作。

——Michael Schrage，MIT媒体实验室，Computerworld

温伯格在全书提供了无数强有力的经历时间考验的见识，这些见识过去是正确的，现在也仍然正确。……如果你在这本书中看不到你自己，你就不是一名计算机专业人员。买一本，读读它，然后把它留在你的经理的椅子上。相信你们俩都将受益匪浅。

——B. Scott Andersen，Amazon.com

ISBN 7-302-07026-1
9 787302 070269 >
定价：39.00元

欢迎访问温伯格专题网站：www.weinbergcn.com

软件与系统思想家温伯格精粹译丛

程序开发心理学（银年纪念版）

[美] 杰拉尔德·温伯格 /著
邓俊辉 /译

B1280654

清华大学出版社
北 京

The Psychology of Computer Programming, Silver anniversary ed.
By Gerald M. Weinberg
EISBN: 0-932633-42-0

Copyright © 1998, 1971 by Dorset House Publishing Co., Inc. All rights reserved.
Translation published by arrangement with Dorset House Publishing Co., Inc.
(www.dorsethouse.com)
through the Chinese Connection Agency, a division of The Yao Enterprise, LLC.

本书中文简体字版由 Dorset House Publishing Co., Inc. 授权清华大学出版社在中国境内独家出版、发行。
未经出版者书面许可，不得以任何方式复制或抄袭本书的任何部分。

北京市版权局著作权合同登记号 图字：01-2002-0810

版权所有，翻印必究。
本书封面贴有清华大学出版社激光防伪标签，无标签者不得销售。

图书在版编目（CIP）数据

程序开发心理学：银年纪念版/[美]温伯格著；邓俊辉译. —北京：清华大学出版社，2003.9
（软件与系统思想家温伯格精粹译丛/张亚勤主编）
书名原文：The Psychology of Computer Programming, Silver Anniversary Edition
ISBN 7-302-07026-1
Ⅰ.程… Ⅱ.①温… ②邓… Ⅲ.软件工程 Ⅳ.TP311.52

中国版本图书馆 CIP 数据核字（2003）第 070761 号

出 版 者：清华大学出版社　　**地　　址**：北京清华大学学研大厦
http://www.tup.com.cn　　**邮　　编**：100084
社 总 机：010-62770175　　**客户服务**：010-62776969
责任编辑：熊妍妍
封面设计：李亚莉
版式设计：李尘工作室
印 刷 者：北京彩艺印刷有限公司
发 行 者：新华书店总店北京发行所
开　　本：175×245　**印张**：26.5　**插页**：2　**字数**：400 千字
版　　次：2003 年 9 月第 1 版　2003 年 9 月第 1 次印刷
书　　号：ISBN 7-302-07026-1/TP·5167
印　　数：1～8000
定　　价：39.00 元

编委会名单

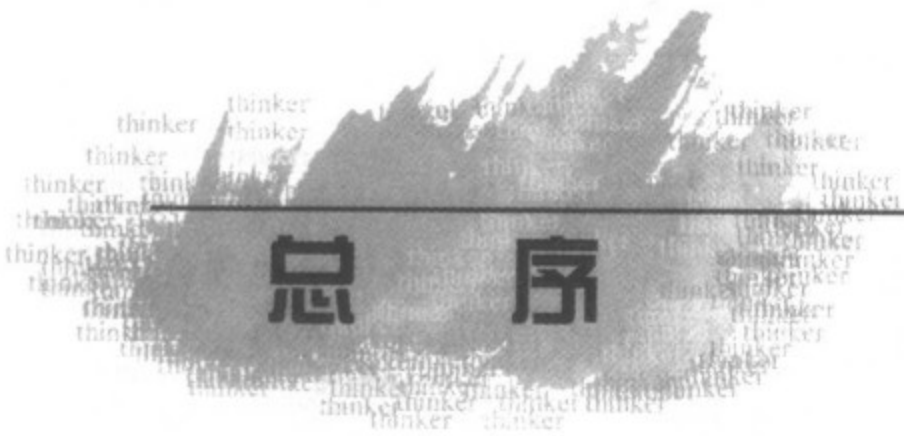
主　编　张亚勤
微软亚洲研究院院长兼首席科学家

顾　问　Gerald M. Weinberg

策　划　熊妍妍

编委会委员（以下按姓氏笔画排序）

万起光　北京湘计立德信息技术有限公司
邓俊辉　清华大学计算机系计算机软件研究所
刘天北　北京奥捷特通信技术有限公司
朱于军　朗讯科技（中国）有限公司贝尔实验室
李　彤　北京诺亚舟管理咨询有限责任公司
杨作兴　北京方舟科技有限公司
周浩宇　创世嘉信企业管理顾问（北京）有限公司
邵维忠　北京大学软件学院
孟迎霞　《程序员》杂志社
章柏幸　北京津诚信业技术有限公司
胡庆培　新加坡国立大学计算机系
曾登高　中国软件网（CSDN）
熊　节　《程序员》杂志社
潘加宇　UMLChina



总 序

Foreword by Ya-Qin Zhang

提到软件开发，人们就会想起微软。常有朋友问我，微软成功的秘密是什么，怎样才能让软件走入千家万户。其实，这类问题早在三十年前就有人完整地阐述和解答过；而且，即使是经历了这么长时间的技术革新，这些论述依然是非常具有借鉴价值和启发性。解答问题的正是这一系列丛书的作者——尊敬的温伯格先生。

温伯格先生是从个体心理、组织行为和企业文化角度研究软件管理和软件工程的权威和代表人物，他有着程序员、系统设计师、咨询师、专业作家的多重身份。温伯格认为：软件的任务是为了解决某一个特定的问题，而软件开发者的任务却需要解决一系列的问题。他自称为“思考着的人”（thinker，而非人们为他定义的“思想家”），同时将他思考的结论和方法通过文字传递给百万计的读者。

温伯格还是一个实干家，他所创建的学校、培训基地，主持的大学、研讨会，给一代又一代软件工作者提供了“清新的空气”。温伯格最喜欢的一句话是中国传统的一句谚语：智者千虑，必有一失；愚者千虑，必有一得。思考是自作聪明者最大的弱项，也是成功者最大的财富。温伯格说，我们不能要求每个人都聪明异常，能够解决所有难题；但是我们必须持续思考，因为只有如此，我们才能明白自己在做什么。

“明白自己在做什么”，听起来是多么平常，具体操作起来却不是那么容易。我想，这可能也是温伯格花那么多心血在著书立说上的原因吧。十年树木，百年树人。温伯格著述颇丰，从早期集中在计算机和软件的实务的作品，到70年代初《程序开发心理学》的出版，写作角度从软件开发实务过渡到更多的人文关怀。以他自己的话说，这些书的目的是为了探讨“人们是如何思考的”。尤其是在后续的《系统化思维导论》、《系统设计的

一般原理》和《你的灯亮着吗?》三本书中体现得尤为明显。

“明白自己在做什么”，是走向成功的必要条件。那些能够很早地领会或感悟到自然发展、社会发展、人类发展、行业发展、软件发展在很长一段时间内的可能趋势的先知先觉者，虽然在这个世界上不到万分之一，但是他们是时代的智者，只要他们愿意去做，他们能够很快地获得成功。他们具有非常敏感的嗅觉和洞察力，能够很好地把握未来几年的软件需求，从而进行应用解决方案的设计、前卫体验理念的构建。或者说，他们能够在行业内把握方向，技术上突破，特别的是在一些尚未发掘的领域异军突起。他们属于时代或行业的领导者，其成功一半是天才，一半是勤奋。

还有一些人，他们对趋势的领会并不十分敏锐，但是他们最大的优点在于能够在经验的基础上踏实前进。他们的成功百分之九十九来自于学习和勤奋的实践。他们是时代和行业的中坚，是事实上的社会的缔造者，当然也是行业的建设者。他们能够很清楚地知道自身的优势和劣势，根据时代和行业的现状，以及自身的经验和积累，进行主流软件开发、生产和实施。他们不一定掌握最新技术，但是他们一般来说资本和经验都非常充足，使他们保持中流砥柱位置的根本在于其能够正确认识到自身和外界的差距或互补，从而调整策略，后来居上或反败为胜。

“明白自己在做什么”，这种态度确保在进行软件开发和研究时保持理性和缜密的思考。经过了十多年的实践，温伯格先生称：“技术是毫无价值的”，我的理解是，如果我们都不知道自己所作所为能给社会或自己带来什么，是根本无法找到那些有价值的技术。而他所说的无用的技术指的恰恰就是那些异想天开、不切合实际的无效劳动罢了。通过和温伯格先生的交谈和我自己在微软工作的经验，我可以负责地说，任何成功者都是其领域内的思考者的人，这种思考，使他们在不知不觉中逐渐向正确的方向转变；而温伯格的这一系列努力，正是让我们进行更深一层次思考的提醒。

我相信不论您是否从事软件开发、研究或管理工作，都能从温伯格先生谆谆的话语中受到启发。

微软亚洲研究院　张亚勤

2003 年 8 月





致中国读者

Foreword by Gerald M. Weinberg

去年，我荣幸地得知，清华大学出版社将要引进出版拙著的一个中译本系列。作为作者，知道自己的作品将要结识成千上万的中国软件工程师、经理、测试员、咨询师以及其他相信技术能给我们带来更加美好的新世界的人们，我感到非常的惊喜。

在我早期的职业生涯中，我编写了大量计算机和软件方面的图书；但是，随着经验的增长，我发现如果我们在技术应用和构建之时对于其人文方面没有给予足够重视，技术就变得毫无价值——甚至是危险的了。于是，我决定在我的作品中加入人文领域的内容，并希望能够给读者带来这方面的思考和重视。

在这之后我出版的第一本书是《程序开发心理学》（*The Psychology of Computer Programming*）。这是一本研究软件开发、测试和维护中的关于人的过程。该书现在已经是银版了（自第一次出版至今已经有 25 年），这充分说明了人们对于理解其工作中人文部分的渴求。很高兴这本《程序开发心理学》也将成为这一中译本系列的其中一本。

清华大学出版社引进翻译的我的系列作品，让我有机会将这些选本看做一个整体来思考，并且发现了其中的一些相通的主题。从我有记忆开始，我就对“人们如何思考”产生了浓厚的兴趣；当我还是一个小男孩时，世界上仅有的计算机常常被人称为“巨型大脑”。我当时就想，如果我搞清楚了这些巨型大脑的“思想”，我或许就可以更深入地了解人们是如何思考的。这就是我为什么一开始就做一个计算机程序员，而后又与计算机打了将近 50 年的交道；我学到了许多关于人们思考的知识，但是目前所知的还远远不够。

我的对于思考的兴趣在这些书中都有体现，而在以下三本中尤其清楚：

即《系统化思维导论》(*An Introduction to General Systems Thinking*，这本书也已经是银版了)；它的姊妹篇《系统设计的一般原理》(*General Principles of Systems Design*，这本书是与我的妻子 Dani 合著的，她是一名人类学者)；一本《你的灯亮着吗？——发现问题的真正所在》(*Are Your Lights On？: How to Figure Out What the Problem Really Is*，这本书是与 Don Gause 合著的)；一本《探索需求》(*Exploring Requirements: Quality before Design*)，这本书也是和 Don 合著的，是有关人们如何思考他们在系统中的真正价值)；还有一本《理解专业程序员》(*Understanding the Professional Programmer*，本书是有关程序员的独特思考过程的)。我对思考的兴趣很自然地延伸到如何帮助他人去清楚思考的方法上，于是我又写作了这个系列中的其他三本书：《咨询的奥秘——成功提出和获得建议的指南》(*The Secrets of Consulting: A Guide to Giving and Getting Advice Successfully*)；《咨询的奥秘——咨询师的百宝箱》(*More Secrets of Consulting: The Consultant's Tool Kit*)；《走查、审查与技术复审手册》(*The Handbook of Walkthroughs, Inspections, and Technical Reviews: Evaluating Programs, Projects, and Products*，这本书已经是第 3 版了)。

随着年龄的增长，我逐渐认识到清晰的思维并不是获取技术成功的充分条件，就算是思维最清楚的人也还需要一些道德和情感方面的领导才能。因此我又写了《成为技术领导者——解决问题的有机方法》(*Becoming a Technical Leader: An Organic Problem-Solving Approach*)。随后我又出版了四卷《质量软件管理》(*Quality Software Management*)，其内容涵盖了系统思维、优先度量、协调行动和变更预期等，所有这些都是技术项目取得成功的关键。

在与各位译者的合作中，通过他们不同的文化视角和层面来审视我的作品，使我的思维和写作都获得了升华。我最大的希望就是这些译本同样也能够帮助你们——我的读者朋友——在你的项目，甚至是整个人生中获取更大的成功。最后，感谢你们的阅读。

杰拉尔德·温伯格

2002 年 10 月 11 日



To Chinese Readers

Last year, I was honored to learn that Tsinghua University Press intended to publish a series of my books in Chinese translations. As an author, I'm thrilled to know that my work will now be within reach of thousands more software engineers, managers, testers, consultants, and other people concerned with using technology to build a new and better world.

Early in my career, I wrote numerous highly technical books on computers and software, but as I gained experience, I learned that technology is worthless—even dangerous—if we don't pay attention to the human aspects of both its use and its construction. I decided to add the human dimension to my work, and bring that dimension to the attention of my readers.

After making that decision, the first book I published was *The Psychology of Computer Programming*, a study of the human processes that enter into the development, testing, and maintenance of software. That book is now in its Silver Anniversary Edition (more than 25 years in print), testifying to the desire of people to understand that human dimension to their work. I was very pleased to learn that *The Psychology of Computer Programming* will be one of the books in this series of Chinese translations.

Having my books translated gives me an opportunity to reflect on them as a collection, and to perceive what themes they have in common. As long as I can recall, I was interested in how people think, and when I was a young boy, the few computers in the world were often referred to as "giant brains." I thought that I might learn more about how people think by studying how these giant brains "thought". That's how I first became a computer programmer, and after almost 50

MAF66/13

years of working with computers, I've learned a lot about how people think — but I still have far more to learn than I already know.

My interest in thinking shows in all of these books, but is especially clear in An Introduction to *General Systems Thinking* (now also in a Silver Anniversary edition); in its companion volume, *General Principles of Systems Design* (written with my wife, Dani, who is an anthropologist); in *Are Your Lights On? : How to Figure Out What the Problem Really Is* (written with Don Gause); in *Enploring Requirements: Quality before Design* (also written with Don, and is abont how people think abone what they really value in systems); and in *Understanding the Professional Programmer* (which is all abont those thinking Processes that are unique to those of us who are programmers). My interest naturally extended to methods of helping other people to think more clearly, which led me to write three other books in the series—*The Secrets of Consulting : A Guide to Giving and Getting Advice Successfully*; *More Secrets of Consulting: The Consultant's Tool Kit*; and the *Handbook of Walkthroughs, Inspections, and Technical Reviews: Evaluating Programs, Projects, and Products* (which is now in its third edition).

But as I grew older, I learned that clear thinking is not the only requirement for success in technology. Even the clearest thinkers require moral and emotional leadership, so I wrote *Becoming a Technical Leader: An Organic Problem-Solving Approach*, followed by my series of four *Quality Software Management* volumes. This series covers Systems Thinking, First-Order Measurement, Congruent Action, and Anticipating Change—all of which are essential for success in technical projects.

I have already begun to improve my own thinking and writing by working with the translators and seeing my work through different cultural eyes and brains. My fondest hope is that these translations will also help you, the reader, become more successful in your projects—and in your entire life. Thank you for reading them.

Gerald M . Weinberg

11 October 2002



译者序

——个体性最后的堡垒

Poetry is what gets lost in translation.

——Robert Frost

译完这本与我自己年龄相近的奇书，掩卷沉思，我的目光再次落在封皮的“Computer Programming”上面。十多年来，这个关键词犹如一个精灵，一直相伴在我的左右：在学生们、程序员们书架上成摞成摞的手册、大全、教程、进阶、技巧、精华、经典、探秘、未公开的技术内幕等等的封皮上，总能见到它；而与本书几乎同时出版的那三卷“The Art of Computer Programming”，不知曾经将包括我在内的多少人引向了计算机科学之路。然而此时此刻，我却觉得这个再熟悉不过的关键词忽然间变得陌生起来，但又不知原因何在。我怀疑这可能就是所谓的“格式塔完形心理”在起作用，为了从其中摆脱出来，我决定尝试一下画家们“从裤裆下看世界”的方法。我打开 Amazon 搜索了一下，发现正在那里热销的书籍之中，与“Computer Programming”有关的有近30 000种（其中最近三年内出版的就有近7 000种），而其中与心理学有关的只有约 30 种。这时再回过头来看看手头的这本书，我终于意识到 Weinberg 所讲的“Computer Programming”的确有其与众不同之处——也许正是这种不同，才使得这本书经过 30 余载风采依旧。

更确切地说，Weinberg 是通过一个不同的视角来看待“Computer Programming”。贯穿本书的一个最重要的思想就是：“Computer Programming”是一项人类活动（Human Activity）。它不再是依靠某个人的技能就可以完成的行为，需要依靠人类集体的相互合作，因此绝不等同于我们通常所说的“编程”或者“掌握编程工具和技巧”。正因为在这个合作的过程中人性已经成为了至关重要的一个因素，所以每个程序员的个

性、人格与其独特的问题求解模式都应该得到尊重。为了使这种合作的绩效达到最佳，需要借助于正确的选拔、培训、组织与管理方法，消除程序员与其主管之间的误解与偏见，并帮助他们从“人之个体性的最后堡垒(the Last Bastion of Individuality)”中走出来。(此外，软件企业与客户之间的矛盾也是一个重要方面，在 Weinberg 后期的“Quality Software Management”系列丛书中对此有深入的讨论。) 时至今日，有很多的程序员和主管依然坚信“编程能力”是与生俱来的；而更可悲的是，仅仅将其视为一门手艺甚至饭碗的还大有人在。面对在软件业界依然十分流行的此类神话，Weinberg 的话值得我们每一个人深思：“优秀的程序员是培养出来的，而不是天生的”(Good Programmers are made, not born.)。但愿更多人能由此猛醒，从中找回作为程序员、作为人的自尊与自信。

Weinberg 对“Computer Programming”的理解与界定也与大学教授们很不一样。后者由于自认为能够站在计算机科学的高度分析和解决问题，总是会不由自主地流露出一种相对的优越感。的确，在计算机科学中所讨论的，都是一些有关计算机的本质与一般性规律的问题，其中有些问题甚至已经近乎玄妙了。因此，在教授们的词典里，“Computer Programming”往往被解释为“程序设计”，他们感兴趣的只是计算机程序在时间与空间上的效率、复杂度和可行性等等。然而 Weinberg 却明确地表示，他并不认为“计算机科学”是一门科学（参见第 12 章评注）。作为计算机科学的一名“忠实信徒”，我当然不敢苟同于 Weinberg 的这一断言。然而我也十分清楚，计算机科学并非十全十美，更谈不上全能全知。一旦将计算机奉若神明，人也就成为了奴隶。在不知不觉中，它成为了人与人之间的一座壁垒，将人与人之间的关系简单而生硬地切断开来。

有个问题一直萦绕在我的头脑中：中国并不缺少（至少是潜在的）精于编程之道的人才，也不乏有志于投身计算机科学的青年人，但是为什么我们的软件研究与软件产业不能取得长足的发展呢？除了人才、资本、市场、法律和体制等因素之外，难道本书中提到的人性不也是一个重要的因素吗？事实上，不同学科、不同领域，总是那样泾渭分明，其间似乎到处都是无形的壁垒。这种心理上的壁垒，也许就是我们最大的阻碍或敌人。

为适应现代计算机科技的发展，国内众多的院校正在进行相应的结构调



整，然而在此过程中所遇到的阻力之大，当事者始料未及，局外人更是难以想象。究其原因，仁者见仁，智者见智，让人莫衷一是，对症下药就更是无从谈起。当然，有一点是共同的——似乎大家都想当然地认为，问题完全来自外部，也就是说，与个人无关。无论如何，这类解释是难以令人信服的（尽管可以令解释者自己信服）。既然如此，我们是否应当将注意力转向我们共同的盲点，比如说，尝试一下 Weinberg 的建议——各自在内心中，自省一番？我相信，虽然不见得这样就能够一劳永逸地解决问题，但也绝不会徒劳无功。或许，我们可以从中悟出点什么——因为，即使你不相信这就是问题的根源所在，但你也很难否认这个方面的确曾经被我们忽视过，忽视过很久，而它却很重要。

钱钟书所揭示的“围城效应”之所以如此深刻，也许正是由于它反映了人作为社会性动物的行为特征。绝大多数人不相信，有朝一日人类能够完全超出动物的层次。然而不容否定的是，计算机这一学科的发展，已经进入了一个全新的阶段，也出现了许多新的问题——比如，在今天，无论是软件产业界还是学术界，一种“反围城现象”屡见不鲜：“城里人既不想逃出去，城外人也不想冲进去”。当然，即使钱钟书先生再世，我们无法从他那里请教来方法，使自己脱离目前的这种窘境和苦境。或许有人会说，关键是要让城里、城外的人不再各自安于自己的现状。然而我却想对这部分朋友说，为什么大家不一同去拆除掉这座城呢？这座壁垒是可以拆除的，而且或早或晚，它必定要被拆除的——因为，它本来就是人为构建起来的。

我喜欢借用金庸先生笔下的华山剑派，来比喻计算机科学与技术这一学科领域；而其中的科学与技术，就犹如“气”、“剑”二宗。二者各立门户，老死不相往来，整个学科就绝不可能取得持续发展，甚至会走向衰落。“他山之石，可以攻玉”，取长补短、相互促进，才是共同发展之道。从这个意义上讲，如果计算机界的同行们能够从本书中获得一点有益的启示，作为译者，我将会感到莫大的欣慰。

就在 2003 年春节联欢晚会中，居然有三个小品分别演绎了父亲与儿子、交警与的哥、医生与病人的角色互换。这也许只是一种巧合，但我更愿意相信，中国人已经开始懂得如何从人性的角度去看待与理解他人，并

通过他人反过来更加深刻地了解自己。但愿这其中有更多的程序员、软件管理人员、软件客户以及计算机科学工作者——未来计算机界的令狐冲，就将出自他们之中。

我不想用“时间仓促”之类的托词，为自己工作中的错误预先铺设好退路。正如 Frost 所言，在翻译这样一本佳作的过程中，其中的“诗意”是注定会遗失的。比如这里的“Computer Programming”一词，最终译作“程序开发”，虽然避免了与“编程”、“程序设计”等混为一谈，却实属反复推敲之后的无奈选择。尽管如此，在向 Frost 定律进行挑战的过程中，我自认已经竭尽了全部所能。因此，我真诚地渴望来自读者的批评指正，你们的意见，将是我发起下一次挑战的信心之源。

在本书的翻译过程中，得到了 Weinberg 先生的多次热情帮助。他对自己的错误毫不避讳或隐瞒，本人曾经因为侥幸发现其中一处小纰漏，得到了 Weinberg 先生寄来的一美元奖励。他对科学的这种严谨而坦率的态度，令人钦佩。

在此，我也想对幕后的审稿人表示感谢，他们为译本的改进与完善提供了宝贵的意见。而他们的细致与认真精神，更是值得我学习的。

我的妻子王霞在繁重的工作与家务之余，也给本书的翻译提供了很多建议，并对整个译稿进行了审阅。再华美的言辞，也不足以表达我对这种支持与鼓励的感激。

邓俊辉

2003 年 2 月 8 日



银版序言

Preface to the Silver Anniversary Edition

《程序开发心理学》是一本令人称奇的书——其令你感到奇特的程度，不亚于你听说有人能够活到125岁时的感受。我们所从事的这一行发展得飞快，据在下所知，还没有其他任何一本计算机方面的图书，在初次出版之后，能够在长达25年的岁月中一直保持活力——而且这种活力到今天仍在继续。

令我自己更为惊讶的一点是，在写作这本书的时候，我根本就没有想到过它会有什么与众不同之处。在那之前的15年内，我一直在编写代码、带领程序员小组以及为程序员们提供培训和咨询服务。我确实曾经考虑过写一部关于计算机程序开发（在那时这项工作还鲜为外人所知）的小说，不过我意识到自己当时的写作技巧实属贫乏，写出来也不见得有人会信。我于1969年赴意大利休假八周，其间灵感突发，于是写成了《程序开发心理学》的初稿。

那时，我已经出版过几本关于如何为不同计算机开发程序的书，非常畅销；但是这本关于**心理学**的书对我来说却是一次新的探险。为了这本书的出版，我经历了许多挫折——有些是我在此前出版书籍时所不曾经历过的。截至当时，McGraw-Hill一直是我的出版商。这家出版社一开始请了几位审阅人，他们看过我的手稿之后一致认定，没有人会愿意买这本书。于是我又转向Prentice-Hall，他们勉强同意——但前提是，我必须同时把我的几本很赚钱的书也交由他们出版。然而我觉得还是找一家更为热情的出版社为好，因此又将书稿同时交给四家出版社进行评阅。到这时，时间已经过去了两年，因此我决定哪家出版社首先表示愿意出版这本书，我就交给它出版。最后四家出版社都表示愿意，但是其中最早的一家是Van Nostrand Reinhold出版社——在1971年，这本书终于付梓出版了。具有

讽刺意味的是，就在这本书出版的当天，负责此书的编辑被解雇了，理由是他“对计算机出版界一无所知”。

无论 Van Nostrand 出版社对这位编辑如何评价，这本书很快就在技术类书籍中成为佼佼者，先后重印了 20 次，并且在 25 年之后仍然在不断印刷。最后 Van Nostrand 出版社将本书以及他们所有的计算机图书的版权都转让给了另外一家大出版社——在此后的数月之内，这本书就因为印刷中断而脱销。经过了几乎是遥无尽头的艰难谈判，我终于重新获得了版权，并委托 Dorset House 出版社来出版本书的银年纪念版[①]。

我之所以要出版这本银年纪念版，是基于以下几个方面的考虑：

1. 让软件行业的新一代人士可以读到本书的原版；

2. 这一领域依然很年轻，我希望能够从历史发展的角度，为这一领域提供一些其所亟需的纵览性观点；

3. 利用这个千载难逢的机会，对软件产业的发展作一评述，并对我以前的思想做一反思。

在这个新版本中，我并不想在软件心理学方面进行更新。首先，Ben Shneiderman 等人已经在这方面颇有建树，其水平远在鄙人之上。其次，正如 Ben 曾经评价的，实际上这本书谈论得更多的并不是软件**心理学**，而是软件**人类学**——在后续的几本书中，我在这两个方面都做了展开论述。

在重新审视本书时我的感受颇多，我认为至关重要的一点是，我希望能够暂时驻足片刻，以体会一下在过去的 25 年间，我自己以及软件产业究竟前进了多远。对于软件产业而言，本书的第 1 版也许不能称作为一个转折点；但是对于我自己而言，它的确是一个转折点。从那时起，我编写的代码少了很多，而且也更少参与对程序员团队的管理工作。然而另一方面，我却为成千上万的程序员及其团队的主管做了培训，并为成百上千的软件项目提供了咨询。我开始把更多的精力用于审查代码、设计方法、审查设计方案、了解需求以及汇总需求。与此同时，我还专门花了很多时间，去培训那些有可能成为软件经理的人，并为他们提供咨询。有的时候我也确实会想要回到从前的生活方式，那种整天惟一的工作就是编写代码

① Silver Anniversary Edition，银年纪念版，即 25 周年纪念版。——译者注

的生活方式非常简单，不像现在这样需要兼顾这么多的其他事务。在软件产业界中，这是一种非常普遍的感受；但是尽管如此，却很少会有人去按照自己的真实意愿身体力行。

我在今天可以明确看出的一点是，这本书的第 1 版实际上就是一张关于我自己所从事的研究方向（比如团队、领导艺术及方法、问题求解以及问题定义等）的领航图。我把此后 20 多年的时光，都用于将这些大题目具体化，因为这些问题一方面非常重要，另一方面却没有很好地为人们所理解。纵览自己在本书之后出版的一系列其他书籍，我觉得自己的确在致力于填补这些“空洞”。请允许我在此按照出版时间列出这些书的名字：

1973：“Structured Programming in PL/C，An Abecedarian”[1]。针对程序开发技术以及编写程序开发文本的新式教学方法，我们进行过一项实验。这本书中汇集了这项实验的成果。

1975：“An Introduction to General Systems Thinking”[2]。这本书直接考察了关于系统思想的各种深入思考过程。（经过 20 多年之后，这本书仍在重发行。）

1975：“Structured Programming”[3]。Dennis Geller、Tom Plum 和我一起写的这本书，荣获了结构化程序设计系列丛书奖。这本书再次就程序开发思想及其教学的新方法进行了探讨。

1976：“High-Level COBOL Programming”[4]。写作这本书的目的，在于帮助 COBOL 的程序员转变思维方式。但是，这一目标似乎并未实现。

1977：“The Ethnotechnical Review Handbook”。Daniel Freedman 和我自掏腰包，出版了这本手册的第 1 版。写作这本书的目的，是希望激励人们，使他们在程序开发的每个阶段都去阅读程序，并且使他们学会如何阅读程序。在当时的情况下，面向市场的出版商中没有哪家愿意花钱出版这种技术评述性的图书。（这本书最后的版本仍然在重印发行。[5]）

1977：“Humanized Input：Techniques for Reliable Keyed Input”[6]。这本书是在人性化界面设计方面的一次早期尝试。

1979：“The Ethnotechnical Review Handbook”。这本书后来做了重新修订，增添了我们后来在阅读及分析程序方面所学到的无数内容。

1979：“The Principles of Specification Design：Film Series and

Workbook"[7]。这本书是与 Bob Marcus 合作完成的，它代表了我们在改进问题定义方面的首次突破性成果。

1979："On the Design of Stable Systems"。这本书是与我的妻子 Dani Weinberg 一道完成的。它将思维模式进行外推，以确定为了生存下去系统应该如何设计。（这本书的第 2 版仍在重印发行，书名改为"General Principles of Systems Design"[8]。）

1982："Are Your Lights On? How to Figure Out What the Problem Really Is"[9]。Don Gause 和我合作完成此书，我们在其中针对问题定义问题进行了深入研究，时至今日它依然是关于问题求解的一本畅销书。

1982："Rethinking Systems Analysis & Design"[10]。这本书的着眼点在于系统分析员的思考方法，或者说他们应该如何思考。

1982："Understanding the Professional Programmer"[11]。这本书的着眼点在于程序员的思考方式，或者说他们应该如何思考。

1985："Computer Information Systems: An Introduction to Data Processing"[12]。在这本书中，Dennis Geller 和我一起针对教学方法以及软件工作中的思维过程等问题作了进一步探讨。

1985："The Secrets of Consulting"[13]。这本书针对软件开发人员（以及其他人员）与其客户之间的关系进行了讨论。这本书的创新之处大放异彩，直到今天它依然十分畅销——这足以说明，在咨询工作中存在着某种具有普遍性的原则可循。

1986："Becoming a Technical Leader"[14]。这本书着重讨论了领导方法以及团队等问题，至今仍然销路看好。

1989："Exploring Requirements: Quality Before Design"[15]。这本书由 Don Gause 和我合作完成，我们在其中将问题定义的问题探究到更深的层次。

1991："What Did You Say? The Art of Giving and Receiving Feedback"[16]。这本书是与 Edie 和 Charlie Seashore 合作完成的。在信息往返传递时，需要人性化的交互；本书总结了我们在这方面已经掌握的知识。

1991－1997："Quality Software Management"[17]。这是一套四卷的系



列丛书。其中囊括了在软件成果以及软件人员的管理方面本人所了解的所有内容。通过这一丛书，我也在系统的思想、测量、行为以及变化等领域，从心理学的角度提出了新的课题。

直到完成了“Quality Software Management”系列丛书之后，我才感到自己在四分之一个世纪之前开始着手研究的问题总算是有了一个总结。回顾这些年的发展过程，尤其是在重读本书的第 1 版时，我清楚地看到原来我是一个比自己预料中还要蹩脚的预言家；但是值得表扬是，我已经料到自己是一个比预料的还要蹩脚的预言家。实际上，这个问题和所有的技术评论一样——虽然比自己预料的要差一些，但是比自己一直担心的情形还是要好一些。

在针对他人工作的技术评论中，我需给人家提一点忠告，对于本书的银年纪念版，我也决定采纳自己提出的这个忠告，那就是：决不要去掩盖自己的错误——因为也许正是通过这些错误，我的读者才会获得更大的收获。为了在这方面对读者有所启发，我力求保持最初版本的固有风貌——不折不扣、货真价实的古董；同时，一旦灵魂受到震撼，我也会增加一些“事后诸葛亮式”的评注。我真诚地希望，你能够透过这个现代历史资料和文物的大杂烩获得某些启示；哪怕是能够对你有些许的帮助，也是鄙人莫大的荣幸与宽慰。

Gerald M. Weinberg

注释：

[1] Gerald M. Weinberg, Norie Yasukawa Freedman, and Robert Marcus, *Structured Programming in PL/C: An Abecedarian* (New York: John Wiley & Sons, 1973).

[2] Gerald M. Weinberg, *An Introduction to General Systems Thinking* (New York: Wiley—Interscience, 1975).

[3] Dennis P. Geller, Naomi Kleid, Thomas W—S. Plum, and Gerald M. Weinberg, *Structured Programming: A Film Series and Workbook* (Kansas City, Mo.: Edutronics Systems, 1975).

[4] Gerald M. Weinberg et al., *High-Level COBOL Programming* (Cambridge, Mass.: Winthrop, 1977).

[5] 这本书后来有所发展，最终定稿为：Daniel P. Freedman and Gerald M. Weinberg' *Handbook of Walkthroughs, Inspections, and Technical Reviews*, 3rd ed. (New York: Dorset House Publishing, 1990).

[6] Tom Glib and Gerald M. Weinberg, *Humanized Input: Techniques for Reliable Keyed Input* (Cambridge, Mass.: Winthrop, 1977).

[7] Robert Marcus and Gerald M. Weinberg, *The Principles of Specification Design: Film Series and Workbook* (Bowie, Md.: Brady Publishers, 1979).

[8] 本书当前的版本是：Gerald M. Weinberg and Daniela Weinberg, *General Principles of Systems Design* (New York: Dorset House Publishing, 1988).

[9] 本书当前的版本是：Donald C. Gause and Gerald M. Weinberg, *Are Your Lights On? How to Figure Out What the Problem Really Is* (New York: Dorset House Publishing, 1990).

[10] 本书当前的版本是：Gerald M. Weinberg, *Rethinking Systems Analysis & Design* (New York: Dorset House Publishing, 1988).

[11] 本书当前的版本是：Gerald M. Weinberg, *Understanding the Professional Programmer* (New York: Dorset House Publishing, 1988).

[12] Gerald M. Weinberg and Dennis P. Geller, *Computer Information Systems: An Introduction to Data Processing* (Boston: Little, Brown, 1985).

[13] Gerald M. Weinberg, *The Secrets of Consulting* (New York: Dorset House Publishing, 1985).

[14] Gerald M. Weinberg, *Becoming a Technical Leader* (New York: Dorset House Publishing, 1986).

[15] Donald C. Gause and Gerald M. Weinberg, *Exploring Requirements: Quality Before Design* (New York: Dorset House Publishing, 1989).

[16] Charles N. Seashore, Edith Whitfield Seashore, and Gerald M. Weinberg, *What Did You Say? The Art of Giving and Receiving Feedback* (Columbia, Md.: Bingham House Books, 1991).

[17] Gerald M. Weinberg, *Quality Software Management, Vol. 1: Systems Thinking* (New York: Dorset House Publishing, 1992); *Quality Software Management, Vol. 2: First-Order Measurement* (New York: Dorset House Publishing, 1993); *Quality Software Management, Vol. 3: Congruent Action* (New York: Dorset House Publishing, 1994); *Quality Software Management, Vol. 4: Anticipating Change* (New York: Dorset House Publishing, 1997).



第1版序言

Preface to the First Edition

写作本书的主要目的只有一个——启动一个新的研究领域：作为一种人类行为的程序开发，或者简而言之，程序开发心理学。所有的其他目标都从属于这个主要目标。例如，在力所能及的情况下，我会尽力增加本书的趣味性，同时淡化技术的味道，从而让更多的人愿意阅读它：这些人不仅包括程序员，也包括程序开发的主管经理；同时，因为我们现在可以通过很多种方式参与到程序开发的过程之中，所以还应包括与程序开发有关的其他所有人。我努力希望实现这样一个目标，即让读者在读完本书之后评价说："嗯，程序开发的确不只是与硬件、软件有关。从今往后，我将从新的角度来看待这些问题。"

正因为这是一个新的领域——或者说是对那些已经为我们所熟知的事物的一种新的观察方法——所以，我们还不能为其中的某些思想找到"科学的"依据。本书中的许多观点，的确仅仅是作者的个人主张（而且是强烈的主张），然而除了本人多年以来的个人观察之外，这些主张尚没有更好的理论基础。毫无疑问，其中很多主张中存在的错误是显而易见的；即使是更有依据的一些思想，也同样可能存在谬误之处。但是我想指出的是，错误的思想与毫无思想之间存在着天壤之别。如果哪位读者对其中的某个观点另有高见，我真诚地希望他能够通过实验来证明我的谬误。

正如我所希望的，书中给出的大量实例说明，大量的误解与曲解阻碍了我们所从事的这个行业的发展；我和我的学生们通过极其简单的实验，已经对此提出了挑战。然而与问题本身的规模之广相比，我们所提供的素材实在是太有限了。各种不同的统计数据都显示出，今天担任程序员工作的人已经有数十万之众。如果我们的经验确实会对大家有所启发，如果程序员及其主管经理在读过本书之后，哪怕只是学会了将程序员当做活生生

的人，而不是另外一台机器来看待，那么这些程序员就将会更高效地发挥作用，同时会对自己的工作更加称心如意。

如果我们学会从心理学的角度看待问题，那么在硬件和软件的设计方面，我认为我们同样有可能取得巨大的进步。但愿软件设计师们在读过本书之后，能够学会运用心理学的原理来指导其设计工作。当然，我并不指望区区一本书中的点滴思想和思索能够为他们直接提供所有的信息；但愿他们能够在本书的启发之下，学会从其他的渠道获取信息。到目前为止，至少从工程或者数学的角度来看，程序开发仍然是一个深不可测的过程；然而实际上，这一过程与心理学有着血脉相连的关系，这种关系如此密切，以至于哪怕是在心理学方面的些许开窍，所带来的帮助都是难以估计的。这个断言并非无稽之谈——我个人以及我的学生们在教学、学习以及程序开发时都会将心理学牢记在心，我们的亲身经历与切身体会都证实了这一点。我希望本书的读者能够通过自己的工作实践，对这个断言进行验证。

本书中显而易见的一个特点就是，其中的思想并非只来自我个人；实际上，这些思想都是在集成了很多人的经验和智慧之后才形成的——这些人中有很多是我的学生，他们来自 IBM 公司分别设在纽约、日内瓦以及纽约州立大学宾厄姆顿分校的系统研究所。在这十年之间，我有幸能够在学生中结识如此之多的第一线程序员；正因为如此，我才能够综合形形色色的程序开发环境的特点，并将自己一孔之见不断扩充、完善——而要是凭借一个人的力量，恐怕任何人毕其一生也不可能有如此丰富的经历。有一点大家应该都能理解，为了尊重（无论是完整的还是有过失的）当事人的隐私，我们将这些经历做了适当的加工整理。

如果没有与学生们长期的相互砥砺，没有他们丰富而且极具启发性的经历，这本书是不可能完成的。另一方面我也觉得，如果出于谦虚的考虑而毫无原则地将书中的一切谬误都归到自己的头上，那也是毫无意义之举；同时毫无疑问地，因为这样或者那样的原因，的确会有一些与我相关的经历会被曲解。然而既然我已经采纳了这些经验，并且用来支持书中的某些观点，那么我就应该承担这部分的责任——类似地，由于读者会根据自己的经验与需求，对书中的所有观点进行权衡、取舍，所以你们应该对



此自行负责。如果说有什么本来应该是我的最后一个愿望，那就是奢望书中提到的某些东西能够为读者接受，甚至成为他们的信条——然而，我并不希望读者在阅读过程中贸然接受某种观点，因为这种读书的态度正是我们应该努力戒除的。本书中提供的素材，只是读者在自己思想形成的过程中需要消化、吸收的食物，而不应该是个人思想的简单替代品。

我想在此感谢我的各位学生、老师以及朋友，感谢他们对我个人的帮助，以及对本书的贡献。老师、朋友和学生——对我来说，他们实际上是一个整体，并且没有区别；我也希望他们这样看待我自己。我要把这本书献给他们，我的这种感激之情不仅限于献辞那一页上，而且已融入于书中的字里行间。我的妻子不仅在日常生活中给了我无微不至的关心，而且还从一位人类学家的视角，阅读了本书的每一页；我对妻子的感情，仅仅通过献辞是远不足以表达的。

Gerald M. Weinberg

第1版序言

评　注

“写作本书的主要目的只有一个——启动一个新的研究领域：作为一种人类行为的程序开发，或者简而言之，程序开发心理学。”(p. 19)

四分之一个世纪之前，序言的开头这样写道。现在有什么变化呢？我在年少轻狂时所定下的目标，是否已经达到？

唉，转眼间已经老了25岁，即使没有变得更聪明一些，至少我的锐气也应该有所收敛。现在确实有很多人已经开始把计算机程序开发当做一种人类行为进行研究，但是我却不敢妄言是因为在下的这一拙作，才开启了这个研究方向的大门——无论它是否真的可以被称为是一个“研究方向”。

如何使用本书

Oscar Wilde[①] 曾经声称，无需在事先做任何准备，他都可以在任何集会场合，就任何话题侃侃而谈至少半个小时。当有一位存心刁难的听众请他就女王发表看法时，Wilde 依然桀骜不驯地答道："但是先生，女王并不是一个话题呀。"

如果谈论的目标根本就称不上是一个话题，那么即使是像 Wilde 这样健谈的人，也无从谈起；但是许多人仍然不相信，程序开发心理学的确是一个话题。毫无疑问，很少有哪个大学开设了这方面的课程，所以要是谁敢于第一个吃这个螃蟹，在大学中开辟这样一门新课程，其艰难之程度可想而知。像本书这样的一本教材，肯定会为这些人提供一些帮助；但是至少对于第一次尝试而言，就如何使用本书向他们提出一些善意的建议，应该不算过分。

这本书是伴随着一门名为"程序开发心理学"的课程而逐步完成的。选修这门课的都是我们"计算机系统"项目中的研究生，他们都具有一定的程序开发经验，并且接受过至少一年的培训。对于那些尚不能独立编写程序、大学尚未毕业，或者虽然主修计算机系统或计算机科学专业，但是还没有达到本科四年级水平的人来说，选修这门课并非明智之举。

本课程是按照学术研讨会的形式进行的——因为在下自认为，如果学生已经懂得如何去阅读，课堂讲授完全就是多余的了。本课程跨越两个学期，每学期各两个学分——这样安排的目的在于，可以使学生拥有尽可能多的时间，以积累个人经验。课程要求每位学生设计并完成一个小型实

① Oscar Wilde，奥斯卡·王尔德（1854－1900）。19 世纪后半期欧洲文学史上的著名作家。这位文学才子、诗人、戏剧家在文学上所取得的成就，不论是生前还是死后，都是举世公认的；同时，他的同性恋问题也引起了人们的很大争议。——译者注

验；其中的一些实验对本书有过很大贡献（在书中可以读到有关的介绍）。选课的学生每星期开一次研讨会，每次两个多小时；事先他们需要阅读有关章节，同时根据对应的参考书目，就特定的讨论主题做好准备。

关于课外阅读，有一项很好的建议。由于有些参考资料属于内部出版，或者是不公开的技术报告，所以我们经常会事先为学生们提供这些阅读资料的复印件。这种方式似乎可以使学生更有规律地进行课外阅读。

我们在进入有关“如何研究程序开发”这一主题时，有点急于求成，因为很少有学生会在行为科学方面有什么基础，更不消说行为科学方面的实验了。只有在这些方面做好准备之后，学生们才具有了着手进行实验的能力，同时我们才能够继续介绍后续的内容。是否要求学生进行实验，本课程没有统一的规定；尽管如此，倘若的确要求做，那么很重要的一点就是：要让学生尽早动手。

如果不要求或者不希望学生进行实验，那么这门课就正好可以按照一个学期的时间来规划——既可以采用学术研讨会的形式，也可以采用课堂讲授的方式。无论采用何种方式，我都强烈建议每个星期均要花一段时间用于讨论；为了准备这些讨论，可以参考书中各章后所附的问题来确定讨论议题——可以要求学生做简单的笔记，也可以通过书面作业的形式。

对于暑期学院或者是管理培训研讨会等高强度的培训课程，由于其时间只有一到两周，所以这本书相应的用法也有所不同。在这种时间紧迫的情况下，我们只能忍痛割爱地取消实验。然而，还是可以通过一次简单的实验，让学生作为主体，切实体验一下实验的过程。本书中所建议的实验不胜枚举，其中任何一个都适宜于此类教学；但是如果时间限制得很严，就必须在事先做好最充分细致的准备。类似地，由于时间紧迫，必须保证每位学员都能够获得属于自己的一整套阅读材料。最后，为了缓冲学员的学习压力，可以将讨论分散开，安排到每次课堂讲授之后进行。如果学员们已经是富有经验的程序员或程序开发主管或经理，那么讨论就显得特别重要——因为他们在讨论过程当中互相提供的信息，并不比从教员或者教材那里所获得的要少，甚至可能会更多。对于他们来说，这些议题非常重要，而且非常贴近他们的实际工作，所以教员只需稍加点拨和组织，课程就将大获成功。



目 录

第Ⅰ篇 作为人类行为的程序开发

第Ⅰ篇 作为人类行为的程序开发——评注 …… 4

第1章 阅读程序 …… 5

一个例子 …… 6

机器的局限 …… 8

语言的局限 …… 9

程序员的局限 …… 11

历史遗留问题 …… 11

规范 …… 13

小结 …… 13

思考题 …… 14

参考资料 …… 15

第1章 阅读程序——评注 …… 17

第2章 优秀程序的要素 …… 19

技术规范 …… 21

日程计划 …… 24

适应性 …… 24

效率 …… 27

小结 …… 30

思考题 …… 30

参考资料 …… 31

第 2 章　优秀程序的要素——评注 …… 33
第 3 章　如何研究程序设计 …… 35
自省 …… 36
观察 …… 38
实验 …… 41
心理学测量 …… 45
利用行为科学中的数据 …… 49
小结 …… 51
思考题 …… 52
参考资料 …… 53
第 3 章　如何研究程序设计——评注 …… 57

第Ⅱ篇　作为社会行为的程序开发

第Ⅱ篇　作为社会行为的程序开发——评注 …… 64
第 4 章　程序开发组 …… 67
正式与非正式组织机构 …… 67
自然环境与社会结构 …… 71
错误与惟我独尊 …… 73
无私式程序开发 …… 78
建立与维护程序开发的环境 …… 83
小结 …… 88
思考题 …… 88
参考资料 …… 90
第 4 章　程序开发组——评注 …… 92
第 5 章　程序开发团队 …… 97
团队的组建 …… 97
目标的设定与认同 …… 103
团队的领导者及其领导方法 …… 109
团队中可能出现的危机 …… 118

小结 …… 125
思考题 …… 126
参考资料 …… 128
第 5 章 程序开发团队——评注 …… 130
第 6 章 程序开发项目 …… 135
调整过程的稳定性 …… 135
绩效评价 …… 140
项目结构 …… 147
大型项目中共同的社会性问题 …… 151
小结 …… 155
思考题 …… 157
参考资料 …… 159
第 6 章 程序开发项目——评注 …… 161

第Ⅲ篇 作为个人行为的程序开发

第Ⅲ篇 作为个人行为的程序开发——评注 …… 169
第 7 章 程序开发任务的差异 …… 171
专业的与业余的程序开发 …… 171
程序员想要做什么 …… 176
程序开发工作的不同阶段 …… 184
小结 …… 190
思考题 …… 191
参考资料 …… 193
第 7 章 程序开发任务的差异——评注 …… 195
第 8 章 人格因素 …… 197
疯狂的轰炸机 …… 197
人格的改变 …… 199
人格中经久不变的部分 …… 202
关键的人格特征 …… 205

人格测试 …… 211
程序员的人格测验 …… 215
小结 …… 217
思考题 …… 218
参考资料 …… 219
第 8 章　人格因素——评注 …… 221
第 9 章　智力水平以及问题求解能力 …… 223
心理集合 …… 223
问题求解的一些维度 …… 227
程序开发的智力因素 …… 230
智力测验 …… 234
程序员的智力测验 …… 236
小结 …… 242
思考题 …… 243
参考资料 …… 244
第 9 章　智力水平以及问题求解能力——评注 …… 248
第 10 章　积极性、培训以及经验 …… 251
积极性 …… 252
培训、课程学习与教育 …… 257
学习的阻力 …… 261
如何学习程序开发 …… 267
小结 …… 273
思考题 …… 274
参考资料 …… 276
第 10 章　积极性、培训以及经验——评注 …… 278

第Ⅳ篇　程序开发工具

第Ⅳ篇　程序开发工具——评注 …… 285
第 11 章　程序语言 …… 287

程序语言与自然语言 …… 287
程序语言设计 …… 292
小结 …… 295
思考题 …… 296
参考资料 …… 296
第 11 章 程序语言——评注 …… 299
第 12 章 程序语言设计的一些原则 …… 301
一致性 …… 301
简洁性 …… 308
局部性与线性 …… 314
传统与革新 …… 318
专用的、多用的以及玩具式语言 …… 325
小结 …… 329
思考题 …… 331
参考资料 …… 333
第 12 章 程序语言设计的一些原则——评注 …… 336
第 13 章 其他的程序开发工具 …… 339
程序测试工具 …… 339
操作系统 …… 345
分时与批处理 …… 353
文档管理 …… 357
小结 …… 367
思考题 …… 368
参考资料 …… 370
第 13 章 其他的程序开发工具——评注 …… 374

第 V 篇 结 语

第 V 篇 结语——评注 …… 385

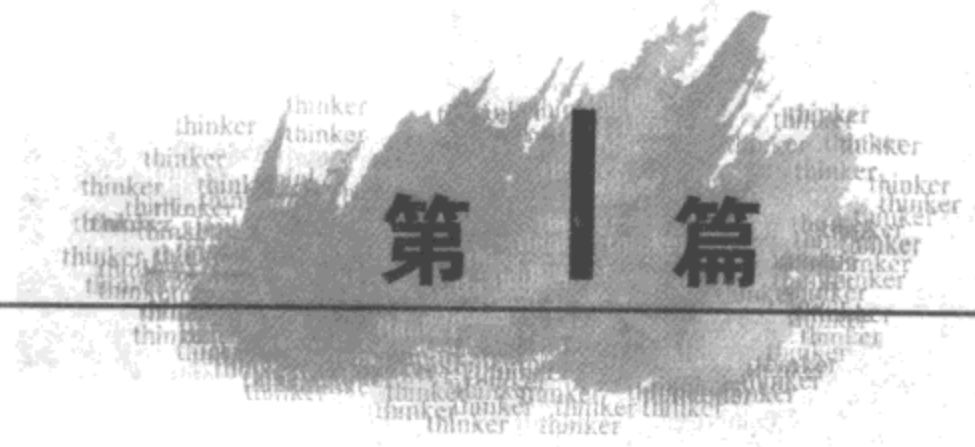
Chinker

第 I 篇

程序开发心理学

Computer Programming Psychology

第 1 章　阅读程序
第 2 章　优秀程序的要素
第 3 章　如何研究程序设计



第1篇

作为人类行为的程序开发

重要的是不要停止怀疑。人怀有好奇心是天经地义、自然而然的。面对永恒性、生命以及令人叹为观止的自然造化，每当陷入关于这些神秘性的沉思时，一种敬畏之情总会在我们的心头油然而生。每一天，即便有幸能够领悟到这种神秘性中的一小部分，对我们来说就已足够。所以，千万不要丧失这份神圣的好奇心。

——阿尔伯特·爱因斯坦[①]

① 摘自："Death of Genius", by William Miller, *LIFE Mag* ⓐ *zine*, May 2, 1995 (c) 1995 Time Inc.

计算机程序开发是一项人类活动。然而，在对程序开发这一过程进行考察时，这个几乎毋庸置疑的断言，却是许多人——许多程序员——共同的盲点。这也许是因为，计算机在这一过程中的作用被过分夸大了。在程序员们的圈内，流行着一个神话——只要有人试图去探究程序开发的进行方式，这个神话就会站出来说不。这些人会告诉你说：程序开发**就是**程序开发，没有定法可循。一个人要么会写程序，要么不会；有的人具备这种能力，而有的人却不具备。

根据这个神话，会很自然地得出一个推论：那些“具备”这种能力的人，应该获得高额的工资——甚至，尽管有些人现在还不具备这种能力，只要他们有可能获得这种能力，就也应该享受高额的工资。计算机管理人员并不情愿支付如此高额工资；他们对这个神话不明就里，却又不得不听任它的摆布，这是他们难以忍受的。或许正因为如此，这些管理人员的心里一直都很清楚：在程序开发过程中，人也**是**一个要素。当然，他们所关心的，往往并不是如何去理解人的这种要素，而是如何去消除这一要素。

多年以来，管理人员一直都在想方设法裁减程序员，他们为此而消耗的资金，数量之大令人瞠目。他们提出了一打接一打的精简方案，为了推销这些方案，不仅本身需要大量的资金，而且还要用种种溢美之词来包装一个承诺：（借助于技术而不是人，就）可以从销售方案，直接到达可行的数据处理系统——然而事实上，这个承诺从来也没有兑现过。管理人员如此高估技术因素（相对于人的因素）所具有的优势，的确是幼稚得可爱，然而，我们却不应该因此而责备他们。恰恰相反，我们应该为他们喝彩，因为他们已经触及到了我们所面临的种种问题的共同根源——无论如何，这是一个进步。管理人员对技术的魔力如此执著，我们当中那些成天被程序员的沉重负担压得喘不过气来的人，或许可以从他们的这份执著中得到启发。管理人员的美梦注定无法成真；尽管如此，他们的这种梦想，却或许可以让我们暂且放下手头的工作，从键盘和终端上解脱出来，然后开始从人的角度，来考察这项以我们自己为主体的、人类的活动。

第Ⅰ篇　作为人类行为的程序开发

评　　注

“多年以来，管理人员一直都在想方设法裁减程序员，他们为此而消耗的资金，数量之大令人瞠目。”(p. 3)

经过25年，在这方面的惟一变化只有一点：为了精简程序员的规模，今天的管理人员需要花费的资金数目之大更加令人瞠目，而且已经远远超出了我当年所想像的数目。时至今日我终于意识到，管理人员的这种愿望是如此强烈并且带有情绪化色彩，以至于他们居然对下面两个事实熟视无睹：

1. 所有精简程序员规模的计划，都是以失败告终的（和以前相比，我们今天在这方面做的工作至少已经增加了10倍）；

2. 所有这些计划的炮制者不是别人，而恰恰正是管理人员热衷于裁减的那些程序员们自己。

虽然人们总是说程序员们缺乏人际交往能力，但是显然这些人擅长于说服别人。就其在这方面的能力而言，他们甚至不亚于已故的、伟大的P. T. Barnum[①]——后者之所以闻名，是因为他毕其一生的亲身经历，用事实雄辩地证明了这样的一个理论：“每时每刻都有一个新的冤大头在上当”(There's a sucker born every minute)。

① P. T. Barnum，巴纳姆（1810—1891），美国知名的马戏团创办人。巴纳姆从出售畸形动物和野兽的生意中赚取了200多万美元。他曾声称在他的动物场里几乎“每分钟都有一只小动物问世”，可后来却常因投资不慎而被人耻笑。——译者注

第1章 阅读程序

曾几何时，人们一度对 COBOL 寄予过厚望[①]；生活在当年的人们会听到，周围人都在大谈特谈管理人员能够阅读程序代码的可能性。回顾这段历史，今天的人们不难理解：当时的管理人员总是急于摆脱受制于程序员的处境，而鼓吹 COBOL 的那些人恰恰利用了管理人员的这种心理，这样就可以从他们那里套取更多的经费。再天真的人也不可能真正相信，有朝一日管理人员将能够阅读程序。这怎么可能呢？要知道，即便是程序员自己，也从不阅读程序。

那么这岂不是说，只有计算机才会去阅读程序吗？换而言之，程序岂不只是为计算机而编写的吗？答案是：既对也不对。我们也许对程序的修改或程序之间的接口等问题不甚关注，但是即使是从学习程序开发的角度看，阅读程序也不无益处。

程序开发也是写作的一种形式，它和其他的写作形式没有什么两样。要学习写作，最直接的途径就是提笔写作；但是对其他的写作形式而言，阅读也是学习的一种重要途径。我们通过阅读（无论是优秀的，还是蹩脚的）文章，也会对学习写作有所促进。但是，究竟有多少程序员会通过阅

① 原文为“... COBOL was the great white programming hope.”，这里的“Great White Hope”指的是 Jack Johnson（1878/03/03－1946/06/10）。1908 年 12 月 26 日，Johnson 在悉尼击败 Tommy Burns，成为本世纪第一位黑人世界重量级冠军。Johnson 不仅将 Burns 打得一败涂地、惨不忍睹，而且多有不恭之辞，在 20 世纪初种族歧视猖獗的美国，这些都注定了其悲剧性的命运。Jack Johnson 的冠军头衔不为当时的白人承认，而白人至上主义者更是竭力寻找能够击败 Johnson 的人，维护白人的“尊严”——这种人被称为“白人的希望”（the Great White Hope）。本书动笔于 1969 年，出版于 1971 年，而 1970 发行的影片“The Great White Hope”就是以 Johnson 的生平为素材的。——译者注

读程序来学习编写程序呢？只有个别的如此，但是绝对不能称为不多。随着终端的出现，情况变得更加糟糕，因为即使是程序员自己编写的程序，也没有哪种形式适合于阅读。虽然现代“计算”技术的历史并不长，但是在其诞生之初，操作计算机并非一件易事。当时，如果要通过计算机的实际运行结果进行学习，所需要的解题时间之长简直令人无法忍受。计算任务的运转周期极其漫长，以至于程序员们只能通过互相阅读各自的程序来打发这段难熬的时光。而另外一些人甚至会利用这段时间，到很远的程序库（当时还是传统意义上的）中去借阅程序。

唉！时过境迁、今非昔比呀。电视机出现后，年轻人很快就抛弃了老式的读书消遣方式；类似地，随着终端的出现以及解题时间的日益缩短，如果谁还有阅读程序的习惯，那么他肯定是一位不可救药的、老套的程序员。夜深人静之时，头发斑白的老程序员会蜷缩在床上，欣赏一段精彩的子程序，或者苦思冥想一个绝妙的宏结构；而那些年少轻狂的程序员，则可能正在通过终端与别人聊得不亦乐乎。毫无疑问，较之沉浸在对他人作品静静的沉思之中，面对面地使用巨型计算机更加令人感觉过瘾。但问题是，后一种方式真的能够使人更加受益吗？

最近，属于我们这个时代的一位青年作家曾经被问及，谁是他最喜爱的作家。他回答得很干脆，自己从来也没有读过小说，因为他认为阅读简直就是浪费时光——对周围的人来说，他这种孤芳自赏的态度简直太新潮了。正如你所估计的，即使从其作品来看，他的这一论调也实在难以站得住脚。或许对于我们那些爱走极端的年轻程序员来说，情况也是如此。在阅读出自别人之手的程序时，我们或许可以从中获得一些收获——哪怕只是从别人的错误之中得到乐趣。如果你希望了解程序员们开发程序的过程，或者说是希望揭开程序开发的神秘面纱，那么只要首先懂得通过阅读程序可以了解到什么，或许我们就找到了开启宝藏的钥匙。

一个例子

为了说明阅读程序所带来的愉悦与成效，让我们来看一个小小的例子。该例子是某个专业程序阅读过程的一个片断，它只是成千上万个程序

中的一段。假设我们碰到如图 1 - 1 所示的一段 PL/I 程序，我们应该如何来解读它？

```
XXX: PROCEDURE OPTIONS(MAIN);
     DECLARE B(1000) FIXED(7,2),
             C FIXED(11,2),
             (I, J) FIXED BINARY;
     C = 0;
     DO I = 1 TO 10;
        GET LIST((B(J) DO J = 1 TO 1000));
        DO J = 1 TO 1000;
           C = C + B(J);
           END;
     PUT LIST('SUM IS ', C);
     END XXX;
```

图 1 - 1　一段待读的程序

我们首先需要一种阅读程序的方法——这与阅读小说不同，阅读程序的最好方法并非总是依次从头到尾。程序不像谜语，我们无法从末尾几页找到答案；也不像一本引人入胜的书籍，只需直接翻到褶皱最多的那几页，我们就能找到最精彩的片断。是的，程序中最好的部分可能出现的位置并不确定——虽然后面我们会学习到，为了完成诸如调试和优化等工作，应该如何找到这些关键的部分。相反地，我们将探寻各部分代码的**来龙去脉**，进而形成一个虚拟的（程序）结构，并以此为基础来进行阅读。换而言之，我们将逐一考察各段代码，并不断问自己同一个问题："这段代码在这里有何作用？"

机器的局限

在解决特定的问题时，和理想计算机相比，现实中运行程序的实际计算机，总是会在某些方面有所局限。正因为这个以及其他的原因，我们才需要对代码做分段处理。在我们的例子中，计算机总共需要读入10 000个数，然后计算其总和；然而，这些数却是按照每1 000个为单位逐批读入的。所有的数都是明确地按照 PL/I LIST 格式打孔表示的；除非由于整个存储器的容量有限，以至于无法同时放下这10 000个数，否则就没有任何道理进行上述划分。换而言之，只有在存储容量少于40 000个字节的情况下，程序员才不得不将求总和的工作划分为多个更小的任务，这时代码中需要多出一层循环；但是一旦存储器的容量很大，足以同时存放全部的10 000个数，那么他也许应该按照与图1－2类似的方式来重新编写这段程序。

```
XXX: PROCEDURE OPTIONS(MAIN);
     DECLARE A(10000) FIXED(7,2),
             C FIXED(11,2),
             J FIXED BINARY;
     C = 0;
     GET LIST((A(J) DO J = 1 TO 10000));
     DO J = 1 TO 10000;
        C = C + A(J);
        END;
     PUT LIST('SUM IS ', C);
     END XXX;
```

图 1－2　消除存储限制之后

当然，即使是为了克服计算机的某种局限，程序员需要加入一些东西时，他也难得会明确地在那里做上相应的说明性标记。虽然这种省略可能会提高程序阅读者的兴趣，但是程序员迟早会因此受到惩罚——比如，在需要把这段代码移植到另一台计算机上去时。最初的程序员可能根本没有

意识到，自己编写的那部分代码是为了弥补计算机本身的某种限制——这种情况下，很难指望这位程序员会相应地做什么标记。比如，为了补偿不同计算机在计算精度上的局限，需要增加很多对应的代码——更准确地讲，我们使用的任何计算机都不是对真正的实数进行计算，充其量只不过是有理数的一个有限的子集。但是程序员们却很容易忘记这些来自硬件的限制，而是把这些理想的假设当做自然而然的。如果我们在程序语言中引入诸如 REAL 等关键词，只会把问题搞得更加复杂——这方面的差异将给程序员带来多大的麻烦，计算机硬件的设计人员则更不可能意识到。

另一个例子是中间存储器，计算机在这方面的局限更加普遍。首先，如果我们制造主存储器的成本足够低，中间存储器就根本没有必要存在。但这只是个理想的假设，实际上，我们不得不使用磁鼓、磁盘以及磁带等存储介质——所有这些都将带来大量的程序开发工作。另外，每种设备都有其特定的时钟特性、定址模式以及存储容量。我们发现大量代码的作用，只是为了克服那些我们将可能遇到的硬件配置（从程序员的观点来看）的不尽完美之处。

语言的局限

因为我们正在探寻程序开发中人所发挥的支撑性作用，所以在这里我们将首先关注的，并不是来自计算机硬件的局限。就人的这种基础性作用而言，如果（实际已是如此）不再有人直接用机器语言进行程序设计，那么程序语言的作用将会更为直接。从机器语言上升到高级语言的结果之一就是，用户可能无法控制硬件的底层功能。一些 FORTRAN 系统中使用的文件尾（end-of-file）判断条件，就是这样的一个例子。在这些系统中，计算机可以准确地识别这些标志，并进而将控制权转交给另一段程序代码；但是 FORTRAN 语言本身并没有提供对应的机制。因此，程序员们只好求助于特定的设备（比如文件尾标志卡片）——在这种情况下，为了克服程序语言的局限性，代码和数据都必须做改动。

在我们所举的例子中，来自语言的局限更加微妙。程序员可以调用 PL/I 提供的名为 SUM 的内建函数，来计算一个数组中若干元素的总和。

但是根据该语言的最初定义，在编写该程序时，SUM 是一个所谓的“数学”函数，而并非一个“算术”函数。这就意味着，该函数假设其输入都是浮点数——如有必要，它将把输入转换为浮点数。即使暂且不考虑效率，如果像上面那个例子，数组的元素是包含小数位的定点十进制数，这种转换也会造成精度上的误差。在使用 SUM 做完统计并结算账目时，虽然账面上的出入可能只有几分钱，但是区区这几分钱却会令很多程序员伤透脑筋。由此引起的不满，迫使人们重新将 SUM 定义为算术类型；但与此同时，在解决类似上述的问题时，却不能再使用该函数了。如果可以去除程序语言上的这种局限，上述程序就可以简化为如图 1－3 的样子。

```
XXX:  PROCEDURE;
      DECLARE A(10000) FIXED(7,2),
              J FIXED BINARY;
      GET LIST((A(J) DO J = 1 TO 10000));
      PUT LIST('SUM IS ', SUM(A));
      END XXX;
```

图 1－3　通过算术类函数求和

在图 1－3 中，PROCEDURE 声明中的 OPTIONS（MAIN）也去除了，这揭示出程序语言中存在的另一种形式的局限——对特定实现的限制。某些系统之所以需要此类拙劣的特性，是为了能够与操作系统进行交互——因为操作系统对主程序 MAIN 的处理与其他子程序不尽相同。

在其他的许多语言中，同样充斥着对旁支末节的无聊限制，而PL/I则几乎没有此类问题，因此也当然会引人注目。让我们看看众多例子中的一个——FORTRAN 中存在的大量冗余代码：有的是因为 DO 循环无法逆向计数，有的是因为不允许用表达式作为迭代的增量或上下限，有的是因为数组下标必须从 1 开始，或是下标表达式的数值不能很大。即便如此，除非这些限制被去除了，否则我们可能根本不会意识到有这些限制——正如我们经常要等到恢复健康之后，才会意识到自己已经病了很久。因此很自然地可以预言，未来的语言迟早会让我们觉察到 PL/I 中现在尚未发现的局限。请注意，这是一个心理学的问题——后面我们将

会回到这个问题。

程序员的局限

与心理因素更直接相关的一个问题是：由于未能通晓其使用的计算机、程序语言或他们自己，程序员们到底写了多少冗余的代码？正如我们所举的例子，程序员可能对 PL/I 的数组概念没有彻底掌握，以至于他不知道只要把数组名放在数据列表中，就可以直接获得该数组中的所有元素。一旦他知道了这个（即使 FORTRAN 也提供了的）捷径，他就会用类似图 1-4 中的方式重写上面那段程序。

```
XXX: PROCEDURE;
     DECLARE A(10000) FIXED(7,2);
     GET LIST(A);  PUT LIST('SUM IS ', SUM(A));
END XXX;
```

图 1-4　程序员知识更丰富后

当然，除了不能了解程序语言的所有功能之外，程序员们还可能存在其他方面的局限——其中一个方面就是我们所说的“词汇局限”。例如，程序员可能不知道某些特定的算法，或者不能同时兼顾一个大问题的各个部分，从而不能有效地避免重复劳动。随着讨论的继续，我们还将就此类以及其他局限做深入探讨——因为很显然，这些问题都属于程序开发心理学的范畴。

历史遗留问题

我们经常可以在程序中找到一些实例，它们都可以归于上面介绍的某种类型的局限；但是它们之所以出现，却是由于程序开发的历史原因。比如，一旦函数 SUM 被重新定义为算术类函数，类似图 1-2 中的代码就没有道理继续出现。然而，代码自有其作为代码存在的道理。要是某段程序

运行良好，那么任何人都不大可能对它进行考察，并且仅仅因为函数SUM的定义改变了，就去修改这段程序。因此在多年之后，一个刚上路的程序员在完成修改这段程序的任务时，可能会为自己比该程序的原作者更加了解PL/I而窃喜。鉴于原作者很可能就是他的主管，一种不良的心理更会滋生出来——顺便说一句，这也是在程序开发过程中客观存在、而且我们迟早必须面对的一种心理问题。

某些代码段的最初起源，已经不能用信仰来解释。有这样一个实例：两位程序员在钻研社会安全部的底层代码时，奇怪地发现了一个人为刻意的设计——只要输入卡片的起始编号为几个特定值，而且在特定某列出现字母“A”，这些“A”就会被转换为数值“1”。实际上，这一列上应该只允许数字出现——这是由一个预编辑程序保证的——因此这个问题就更是令人费解了。尽管这两位程序员搞不懂其功能，他们还是执意不肯修改这些代码，并着手进行调查。最后，他们终于找到了答案。

若干年前——在一些软件公司，这意味着跨越了几代程序员（这也是心理学的另一个问题，对吗?）——在提供卡片的一个区办公室，有一台打孔机出现了故障，把在这一列上的数字“1”都误穿成字母“A”。而当时还没有使用预编辑程序，因此这些卡片可能会被传递给内部程序，并造成计算机的意外挂起。等到问题被发现时，已经有数目不详的此类卡片流通出去了——在这种情况下，最简单的办法就是对该段代码进行“临时性”的修改。由于做完这个修补后，问题完全得到解决，因此人们很快将此事忘得一干二净（又一个心理学问题），于是这个问题一直保留下来，直到这两位热衷“考古”的程序员把它挖掘出来。

鉴别古旧的代码，并非总如上面这个例子那般容易。特别地，随着程序规模的增长，在其开发早期的特定历史条件下所选择的方案，其影响会传播得更久远。一个程序的整体结构，决定于最初开发该程序的软件组的大小及其人员组成——因为整个项目需要进行划分，并分配落实到具体个人，而人各有所长，也各有所短。确实，程序开发组的社会组织结构，是我们感兴趣的一个重点方面。

规范

如果将图 1-1 和图 1-4 做一对比，我们也许会开始相信，毕竟只有极少部分的代码会与我们所试图解决的问题有关。无论从任何角度来看，这一看法都客观地反映了实际的情况——尽管在几乎所有的程序中，总可以找到一些代码，现在的确已经成为人们所需要的功能。即使能够成功地从程序中剥离出这些核心部分，我们也不应被假象所误导，以至于认为：可以在一开始就以该内核为规范；至于其他的局限，则可以由其他的系统来应付。在知之甚少的情况下，程序员很难确定其最终的意图；如果在制定规范时，缺乏对计算机功能的起码了解，据此去写出高效的代码也是很困难的。即使抛开这些显而易见的困难，在大多数情况下我们都要面对这样一个事实：只有在飞快地实际动手编程后，我们才能搞清楚到底想要干什么。

规范随着程序和程序员的发展而获得改进。编写程序的过程本身就是一个学习的过程——无论是对程序员本人，还是对程序的委托开发方，皆是如此。而且，这种学习是在特定的计算机硬件上使用特定的程序语言，由特定的程序员或其所在的开发组在特定的工作环境中，以一系列特定的历史事件为背景进行的，这些事件不仅决定了程序的形式，而且包括代码的功能。

从某种意义上讲，之所以要研究人们编写程序的过程，最重要的目的并非是为了提高程序的效率、使之更加紧凑、更加廉价或是更加易于理解。相反地，这样做的最大好处在于有望从代码中提炼出我们真正需要的东西——而不是我们通过摸索式的、笨拙的方法所能够得到的。

小结

程序被编写成什么样子，取决于众多的因素；然而因为习惯于从外部去考察，而不是去阅读程序本身，我们总是无法认识到这些因素的多样性。一旦我们真的阅读了程序，就会发现无论是否必要，其中这些代

码之所以如此编写，有的是由于计算机的局限，有的是由于程序语言的局限，有的是由于程序员的局限，有的是因为历史的偶然，而有的则可能是因为规范。但是，不管究竟是什么原因使最终的软件加入了某段特定的代码，这种原因必然有其基于心理学的一面。这使我们相信：把程序开发作为一项以人为主体的活动来加以研究，将会取得丰硕的成果——虽然其中有些会出乎你的预料。

思考题

请主管回答：

1. 如果你是一线的主管，你是否有能力阅读你手下的程序员所编写的程序？或者，就阅读程序的能力而言，你是否与“老一辈”的计算机和语言一同落伍了？如果你有这个能力，你会真的去阅读程序吗？如果此前你不阅读程序，为什么现在不尝试一下，看看会发现什么？

2. 如果你是高层的主管，你手下的一线主管们是否有能力阅读他们手下的程序员所编写的程序？你能肯定吗？向程序员们了解一下，然后重新回答一下这个问题。然后查明一下一线的主管们是否真的阅读程序，即使他们有这个能力。我们的调查显示，由于各种原因，其中九成一线的主管不阅读程序。你是否认为，如果不经常地阅读程序员的程序，一线主管们可以知道程序员的能力是否强，工作是否出色吗？

请程序员回答：

1. 你上一次阅读别人的代码，是在什么时候？为什么经过了这么长的时间？最近一次有人阅读你的程序并且与你讨论，是在什么时候？这个人是你的主管吗？

2. 到程序库或者向你的好友借一份程序。试着去分析并找出其中各段代码存在的原因，看看是否属于本文中介绍的某种情况。通过这个练习，你有什么收获？

3. 找出你至少一个月之前亲自编写的一个程序，参照第 2 题的方式进行分析。从这个练习中，你又有什么收获？

参考资料

□ Bucholz, Werner, ed., *Planning a Computer System: Project Stretch*, New York, McGraw-Hill, 1962.

计算机的各组成部分之所以会成为其中的一部分，原因不尽相同。计算的机器是如何演化成现在这幅模样的？针对这一问题，这本书的分析论述极为精妙，其见解之精辟，至今还没有哪本书能够出于其右。此外，对于计算机在程序开发方面的缺陷，这本书中也有深刻的阐述。其原因在于，在所有其他的目标中，Stretch 项目的首要目标是要造出一台“适用于程序员的计算机”；正因为如此，有很多我们在设计其他计算机时认为是想当然的原则在这里都被放弃了。

□ *PL/I Language Log*, IBM Corporation, Confidential File.

这份材料不容易获得，但是时至今日，IBM 不再独自负责维护 PL/I 的完整性，因此我们还是有可能读到它的一份复印件的。这份“语言日志”完整地记录了就我们现在所知道的、针对 PL/I 的争论与决策，因而堪称为演示程序语言制定过程的一本绝佳教材。希望其中的一些部分在经过整理之后，有朝一日能够成为一本条理清晰的图书。

□ 由用户群体或计算机制造商维护的任何一个程序库。你们的程序库提供了丰富的信息。请光顾那里吧！

□ Weinberg, G. M., *PL/I Programming: A Manual of Style*, New York, McGraw-Hill, 1970.

程序缘何会被写成现在这个样子？如果你希望了解作者对这个问题的看法，那么就请阅读此书吧。

□ Gruenberger, Fred, ed., *Fourth Generation Computers: User Requirements and Transition*, Englewood Cliffs, N. J., Prentice-Hall, 1970.

在对前三代计算机的优点进行总结的基础上，这本书对“第四代计算机”进行了展

望。在这本论文集中，几位作者最后似乎都殊途同归式地遇到了一个相同的问题：计算机用户在心理方面的需求。从对计算机产品的影响来看，前期的设计师、推销员以及制造商的作用，要比后来的计算机用户更大——关于这个问题，Murray Laver 曾经进行过跟踪分析。关于在系统设计时，需要在软件的层次上更多地为用户考虑，Jacques Bouvard 也进行了讨论。

第1章　阅读程序

评　　注

关于本章所讨论的问题，这些年来并没有很大变化。为了了解我们编写程序的过程，阅读程序仍然是一个关键。出现了一个改进：为学习而设立的作品库——比如类库——不过，这些作品中缺少评价性的注释，否则它们应该是更具价值的培训资料。模式运动[①]致力于解决这个问题，但是依然任重道远。有关组织指出，在转向新技术的过程中出现了一些问题，其中之一就是必须经过很多努力，才能有效地理解特定开发环境中的类库。

一个变化就是代码中保留了其成长过程的历史痕迹。毕竟，经过了超过25年的积累，这些代码中积淀下来的东西太多。从软件维护的角度来看，这些积淀是有害的；但是就把程序开发作为以人为主体的行为来加以研究而言，如果我们能够得到这些历史记录，以及能够解释各段

① Patterns Movement，模式运动。70年代末期，由哈佛大学教授、建筑师 Christopher Alexander 发起并倡导的一场运动。其最初的思想与理念针对的是建筑设计以及城市规划，然而后来在城镇规划中也得到了广泛的应用，但该运动后来趋于平静。1987年起，模式概念又在 OOPSLA（Annual ACM SIGPLAN Conference on Object-Oriented Programming，Systems，Languages，and Applications）中重新被提出。1994年，在软件领域被称为“四人帮”的 Erich Gamma、Richard Helm、Ralph Johnson 与 John Vlissides 合作出版了“Design Patterns Elements of Reusable Object－Oriented Software”一书，将模式的概念与面向对象的体系结构及其开发（Object-Oriented Software Architecture and Development）联系了起来。

在软件领域，模式指的是那些可以令程序员在设计方面实现知识共享的设计方法。借助于模式，我们不仅可以实现代码与知识的复用，也可以使软件系统的结构更易于理解，而且也可以大大减轻文档化工作的负担。这样，每当我们需要解决一个实际问题的时候，都可以直接从现成的模式开始，而不必考虑最底层、最基本的问题。——译者注

代码缘何发展成这个样子的那些文档，那么我们也许会从中获得一些益处。

但是，在软件维护的过程中被阅读得最多的代码，往往并非是最好的代码。也许正因为如此，这部分代码才最需要维护。对于根本区分不出代码优劣的那些用户来说，阅读程序的确不会有什么帮助。25 年来，这些问题始终没有改变。

但是，开发人员把他们的工作提交给别人审阅，或者希望借审阅他人的工作来提高水平时，为什么会如此之难呢？令人奇怪的是，高明的程序员善于通过演练和审查过程来发现有价值的东西，而那些自以为是的人却不是这样。正因为此，和司空见惯的情况一样，高手越来越高，低手越来越低。

第2章 优秀程序的要素

如果准备把程序开发作为一项以人为主体的行为来研究，我们首先就需要确定一些标准，用来衡量程序的性能。换而言之，在谈到某位程序员比另一位高明，或者某个程序比另一个程序更优秀时，我们需要知道所指的是什么。尽管我们对这些问题多少都有些概念，但是我们将发现，答案并不像想像的那么简单。道理很简单：程序开发不仅是一项人的行为，而且是一项人的复杂行为。

请再次考察一下我们举过的例子。如果更仔细地分析图 1-1，我们会发现，其中只有非常小的一部分——充其量不过 20%——与问题本身的要求直接相关。而在其余超过 80%中，即使不是大多数，也有相当多的部分可以删减——通过心理上、而不是肉体上的更新。实际上，即使是存储容量的限制，也可以通过一台适宜的虚拟机来处理；所以计算机本身的局限，可以通过基于心理方面的适当设计加以克服。

图 1-1 中的程序虽然不长，但根据对实际用于解决该特定问题的程序的统计，却是很典型的。由于人们不去阅读程序，所以很少有人能意识到这种共性的普遍存在。这确实很普遍，因此也为我们指明了改进程序开发技能的更多方向。然而，除了统计无关的（至少是无足轻重的）代码的比例外，还有更多衡量优秀程序的标准。确实，仅仅通过程序自身，我们不可能真正对它们进行评估，说出是好还是坏。

多数程序员认为，确实有这样的程序开发之道。虽然这种感觉不无道理，但是并不能由此得出结论：确实存在某种关于好程序的标准。这种标准根本就不存在——我们不可能脱离其成长的、被使用的环境条件，而宣

称某个程序的优秀程度为 83.72%——至少从这个意义上讲，没有这种标准。

请考察图 1-1。它是一个好程序吗？它也许是一个笨拙的程序，但这只是总分的一部分。毕竟，我们是在阅读该程序之后，才发现了它的笨拙；但是对一般的程序，永远不会有人去阅读它们——因此，不能把外观是否好看作为评判一个程序是否优秀的绝对标准。这个程序是否高效？只有在对计算机、编译器以及程序运行时的代价条件有充分的了解后，我们才有可能回答这个问题。它能按时完成吗？它会消耗哪些资源？它的功能是否符合事先的技术规范？只是浏览程序本身，是回答不了这些问题的。

无法找到一种绝对的尺度来衡量程序的好坏，这成为我们研究程序开发心理学的一个障碍。那么，是否可以通过某些相对的标准来衡量呢？比如，我们可以说 A 程序要比 B 程序好（或者坏）吗？很不幸，由于多方面原因，这种办法一般也行不通。首先，什么时候才有另一个程序用来做比较呢？而就算有这种对比的可能，比如评价一个 FORTRAN 编译器，我们又应该如何统一其各方面不同性能的重要性呢？比如有两个编译器，其中一个编译本身很快、但生成的代码运行速度慢，另一个编译得很慢、但生成的代码运行速度快，难道可以说前者优于后者吗？

但是，不还是有很多时候，我们确实可以进行这种相对的分级吗？一个可以快速编译生成快速代码的编译器，难道不是比一个不仅编译慢、而且生成的代码也慢的编译器更好吗？应该承认，这种相对的比较有时确实可行，但是我们还是不要草率地进行这种比较。就对编译器的评价而言，除了速度以外，还有很多其他方面，随便列举就有：可调试性、语言的覆盖范围、目标代码的可靠性以及对运行过程的监测等。只有在极少数情况下，我们才能找到一对足够相近的程序，可以进行逐点比较，然后找出在所有方面都更好的一个。

这样，我们测评程序时更经常采用的方法，不是拿它们作对比，而是依照某种（整体性的）具体条件——这就是它们各自的开发环境。只要能够公正地对待这些不同的条件，那么我们就决不会去追求什么“最优秀的”程序，甚至连“优秀的”程序不大会感兴趣，而是把目光放在那种符合特定需求的程序上面。

技术规范

在对程序的所有要求中，首当其冲的要求是它必须正确。换而言之，对于任何一种可能输入，它都应该给出正确的输出。当我们说一个程序“可行”时，指的就是这个意思。我们经常听到“可行的程序总比不可行的要好”，确实如此。

一些人的脑子里成天考虑效率或其他次要问题，有一个例子可以帮助他们更好地理解上面这个道理。曾经有个程序员被请到底特律去参与一个新程序的调试，该程序的作用是，确定用来组装某几部汽车所需的零件。程序的输入是一摞卡片，每一张代表一辆汽车的订单，其配置根据顾客的各自要求而不同。该程序收罗了零件各选项的所有可能要求。例如内部装潢中后座的选择，可能包括表面颜色、表面质地、采用豪华材料还是人造革，以及是否装配空调。其中关于空调的选项，很好地说明了问题的基本复杂性——因为，即便是没有受过任何训练的人也能够看得出，空调与后座的装潢之间没有任何关系，但是这种关系确实可能存在，比如可能要为空调的管道腾出额外的空间。于是一般而言，每个选项都可能对零件的选取有影响，因此如果计算机能自动确定需要的零件，那简直棒极了。

不幸的是，当该程序员到达现场时，解决该问题的基本方法已经确定下来了——而且很糟糕。虽然可能影响到每一个选择方案，每一个选项却在程序中被当成一个单独的判断和分支处理。就某种意义而言，这个程序是一棵（由众多测试和判断组合而成的）硕大无比的树——至少有5 000个分支，每个分支代表一种零件的选择方案。以这种方式合计起来——即使再加上16个程序员一起工作——也不可能进行调试，因为每一种情况都必须分别进行测试。为了测试该程序，需要任意输入一张卡片，然后分析其输出。当上面提到的那位程序员到达时，情况异常糟糕，以至于输入一张通常的卡片都会输出奇怪的汽车配置：比如8个轮胎、没有引擎外加三套内部装饰——简而言之，糟透了。

正如处理程序开发中遇到的其他此类糟糕问题一样，人们并不认为它

糟糕。相反，整个开发组双班倒地查找错误，忙得不亦乐乎；并且，不断地从全国各地调集新的程序员前来支援，其中就包括我们提到的那位。很自然地，这只会把事情弄得越发的糟糕。几天后，我们的程序员终于做出决定：这种方式根本于事无补——无论如何，为此远离家人并昼夜奋战，确实毫无道理。由于其“不合作”的态度，他受到众人的谴责，后来主管终于允许他离开开发组。

在返回的飞机上，他才有机会进行平静的反思——这是近一星期来的第一次反思。很快他就意识到这种方法的错误所在，并且悟出一个更好的方法——将整个工作分成两个部分。主运行程序只要不断地简单循环，依次检查精心设计的一组功能规范表，并且在每次确定方案时，只需对其中的各零件分别进行一次单独的测试。这样，该程序至少能保证输出合理数目的轮胎、引擎以及其他零件。只要按照实质上与工程的技术规范一致的形式写好输入，这些功能规范表本身将能被编译出来。这就允许工程人员——而不必是程序员——来根据技术规范进行核对，同时在改变规范要求中的某个零件后，也可以在沿判定树继续向下前进的过程中，不必改变后续的所有零件。

当下飞机时，他已经把这两个程序都编写完了。接下来只用了一天的时间就完成了对程序的检查，再用两天与当地装配厂的工程师一道做好了输入的各技术规范表格。经过在装配厂一周的测试，正当准备返回底特律，并把其成果告诉那里的人们时，他收到一份电报，告知该项目已被撤销——理由是，这个程序根本“不可能”完成。

他匆匆挂了一个电话就登上了飞机，带着他完成的这个程序返回了底特律。在看过演示后，主管们被他说服并相信该项目可以继续，他们要求他给其他程序员做一次报告。很自然地，这些程序员的态度非常冷淡——这又是一个我们将在后面回来讨论的现象——在他讲解其方法的过程中，他们只是静静地坐在那里。甚至到最后，也没有人提出什么问题——直到旧系统最初的创建者举起了手。

“但是**你的**程序需要运行多长时间呢?”他问道——语气的重点放在“你的”上。,

“这取决于输入,”程序员回答，“但是平均而言，每张卡要 10 秒。”

“啊哈，”对方不无得意地说，“要知道，我的程序处理每张卡片只需 1 秒。”

听到这里，所有听众——毕竟，他们都为那个 1 秒的程序付出过“心血”——似乎心里都长出了一口气。但是我们显得年轻而天真的主人公，并没有屈服于这种论调。相反地，他语气平和地说：“但是你的程序不可行。如果不考虑可行性，我可以写出一个程序，每秒钟能处理 100 万张卡片——其速度甚至可以超过读卡器。”

毫无疑问，这些话并不能让他得到来自听众的更多支持。尽管如此，这番话里却包含了对一切程序进行评估时必须依据的一条根本性真理。如果程序根本无法正常运转，对其效率、适应性以及生产成本的评估就毫无意义。无论如何，我们需要务实一些，需要承认：也许根本没有哪个完美的程序曾经被写出来过。每一个真正大型和重要的程序，“都必然包含很多个纰漏”。所以，程序符合其事先制定的技术规范（可行性）的程度不尽相同，在对程序进行评估时，必须考虑到其不完美的一面。

比如，任何编译器迟早都会遇到很多“有毛病的”程序，这时就无法编译通过。但是究竟什么是程序中的“毛病”，在很大程度上取决于人的观点。如果这种毛病出现在你的程序中，你几乎不可能会认为它是毛病，即便其他成千上万的程序员不曾遇到过这个错误。然而如果你正在写一个编译器，那么就要根据遇到问题的用户数量及带来的相应后果对错误进行评估。这项工作的进行并非总是那么科学。确实，那些嗓门大的人以及热衷于给高层领导写信的人，往往会左右评估的结果。但是无论采用哪个系统，程序中的错误总会存在，因此虽然一个编译器可能会令大多数人满意，但是也同时会令个别人不悦。

因此在实际上，为单个用户编写的程序，不同于“软件”的一段。当用户的数量很多时，他们的技术要求也多种多样；如果技术要求本身就五花八门，那么关于什么才是此时可行的程序标准也必然会多元化。在对程序开发的实践进行探讨时，我们不得不考虑到为单个用户和为众多用户开发的程序之间的差别。二者必须分别区别对待，也必须按照不同方式开发。

日程计划

即使不考虑符合技术规范的问题，效率的问题仍然不是最重要的。程序开发中经常遇到的一个问题是要符合开发的日程计划，推迟完成的程序常常没有意义。最起码说，我们不得不比较一下，到底是一个可能（在未来）完成的高效率程序带来的潜在节省更大，还是没有一个程序的损失大。有这样一个值得注意的例子：据一家公司估计，其委托另一家软件公司开发中的线性规划软件，每个月可以为其石油炼制的工序节省 100 万美元。这样，即使是只比开发计划推迟 1 个月完成该软件，该软件也需要在没有任何其他费用的前提下运行 10 年，才能挽回这 1 个月的损失。

因为软件推迟发布而产生的损失并非总是如此严重，但是纵然忽略经济方面的损失，至少在美国，一旦开发计划没有按时完成，总会有一大堆令人烦恼的后果。实际上，一般的开发主管宁可先做 12 个月的计划，然后在 12 个月内完成，也不愿计划为 6 个月，但却花费了 9 个月。虽然在这个问题上，心理学大有可以研究的余地，但是许多情况显示，真正困扰人们的并非预先估计的平均开发时间，而是实际消耗时间的标准偏差。因此，大多数人宁可每天早晨花上固定的 10 分钟等待公交车，也不愿每周有 4 天只用等 1 分钟，而最后一天花 26 分钟。尽管就平均等待时间而言，后一种方案只要 6 分钟，但是由于某次无法预测的长时间等待将打乱计划，这点好处无法弥补其损失。

如果这个结论也适用于程序开发，那么在考察一系列程序的开发工作效果时，就需要衡量开发时间的方差——而不是像目前的有关研究一样，只考虑时间的平均值。在后续的所有讨论中，我们将尝试对这种效果进行估计。

适应性

在讨论完符合功能要求和开发日程计划的问题之后，我们可能会设想，终于轮到讨论下一个问题了：如何根据效率来衡量一个程序好的程度。但是在运行效率和适应性两个因素之间，相对而言哪一个更重要呢？

无需回避这一问题的实质，事实将会证明首先讨论后者更有利。毫无疑问，有些程序确实只被用到一次，然后就被打入冷宫；同样地，有些程序甚至还没来得及被使用，就被扔到一边。虽然如此，被编写（尤其是由专业程序员编写）的程序，绝大部分都有一定的生命期。而且在这段时间内，其中的大多数都要经历修改。

是的，在其生命期内，多数程序都会被修改——无论其经验的多少，绝少有那位程序员能够反驳这一论断。既然如此，为什么在必须修改以前的程序时，我们总是觉得这项任务如此艰巨，以至于往往决定弃之不用，干脆自己从头写起？只要阅读过程序，我们就会透过这些程序发现：实际上，很少有哪位原作者会考虑到可能的后续修改。然而这还不是问题的本质，充其量只是症状。有人也许会问，既然程序员们完全知道他们的程序迟早会被修改，为什么在编写程序时却从不考虑这一问题呢？为什么他们的程序有时甚至会让人觉得，这些程序是被恶意地搞成这副模样的，其目的是为了抵制以后的修改——就像犹如法老们的坟墓对付所有可能的入侵者一样？

这些问题的答案是有的，答案来自我们在心理学方面的研究。但是我们目前的任务，还不是要找到答案，而仅仅是提出这些问题，并在我们对关于好程序的标准进行讨论时，能够意识到其重要性。当然，一个相关的问题就是文档——因为如果不是为了使程序更加易于修改，我们为什么要去为程序编制文档呢？我们都会认为，无论是否在计划之中，文档的质量和进行修改的容易程度，应该在很大程度上决定到对程序（或者编写该程序的程序员）的评分。我们会因此进一步同意：为取得这方面的改进，却是值得在心理学方面进行研究。

但是在转入对效率的讨论之前，我还有一句忠言：适应性不是没有代价的。一点不假，我们常常会遇到一个程序，在其他各个方面都很适应和令人满意——但是通常我们在购买时所希望得到的功能，在付钱后却常常得不到。R. A. Fisher 通过数学方法对遗传系统进行研究后，得到一条定理，我通常称之为“Fisher 基本定理”（Fisher's Fundamental Theorem）——鉴于 Fisher 提出过许多其他的重要定理，从这个定理的名字可以看出其非凡之处。用我们这里的术语来表述，Fisher 基本定理告诉我们：一个系统对某一特定环境的适应性越强，它适应新环境的能力也就越弱。稍微放纵一下我们

的想像力，我们就会同意，该定理也适用于计算机程序开发，正如它适用于蜗牛、果蝇或乌龟一样。

为保证程序的效率，必须利用待解决问题及其所运行于的计算机的特殊性。如果在设计程序时忽略了计算机的结构，就可能在运行代价上受到严重惩罚；而程序员如果能够发现并利用问题的特殊条件，就能够加速处理问题的速度，并降低问题的规模。关于此类程序的开发，有一个特殊、但又不失一般性的例子：有一个汇编程序，功能是通过特殊技术查找待翻译的操作码。程序员注意到——或者，发现——有一个数，当与被解释为数值的操作码位组合相乘时，会得到一系列连续的、毫不相同的地址。通过这个乘法器，他可以简洁而同样快速地完成从符号代码到机器语言的转换。

这名程序员因其构思精巧而得到祝贺——这个例子曾经作为原型，被演示给若干代程序员，让他们懂得应该如何完成他们的工作。然而令人伤心的是，这段代码很短命，因为不久之后计算机的操作码指令系统就做了扩充——这是一种频繁经常发生的事。指令集中增加了约十来条新指令后，原来的乘法器不再具有其简洁性、惟一性的特点。实际上人们发现，新的指令不可能有乘法器能如此运转。

尝试设计这样一个新乘法器的结果，是摒弃这一特定的技术，而采用更具一般性的“哈希表搜索”（Hash Table Search）算法——它允许重复元素的存在。不幸的是，为了在变化了的环境中实现这种老技术，人们消耗了大量的精力，直到最后还是决定放弃它。实践证明这种做法只会浪费时间。检讨过去，在其生命期内，采用这种特定技术所节省下来的时间，根本无法抵消在引入新指令后，为保留它而白白浪费的光阴。如果在一开始就采用更具一般性的哈希表方法，确实在效率上会有一定降低；但是——正是这个“但是”造就了这个例子——后续的修改就完全是举手之劳了。事实上，如果采用这种通用的方法，那么在每次“修改”时，除了将新代码插入到哈希表中合适的位置外，没有任何实质性的修改。

类似这样的例子不胜枚举，但是它们都说明了同一个道理。如果强调的是程序的效率，那么我们往往去追求“紧密式”的代码，而如果在未来要对这些代码进行修改，那将会非常棘手。如果使用的是更高层的语言，那么为了使程序更加高效，我们往往需要深入到机器语言层。这种做法至

少抵消了原本用更高层语言编程的一个好处——在不同机器之间的可移植性。其实际效果是，我们将被局限于特定的某台计算机或特定的某个实现——基于这种实际的感受，我们不得不承认这难以令人满意。

但是那些叫嚣着要追求效率的管理者，正是那些在得知修改代码的沉重代价后，为此而伤透脑筋的人。而相反地，那些过于强调程序的通用性和易于修改性的主管们，在发现自己的程序运行起来又慢又费空间时，往往又会变得怨天尤人。对于这些问题，我们必须保持一种成熟的心态——要知道，无论是心理学还是魔法，都无法帮助我们同时实现这两个相互矛盾的目标。要求程序同时具有高效率和适应性，正如企盼找到一位美丽而谦恭的妻子。尽管美貌与谦逊同时集于一身并非不可能，但通常我们还是必须在二者之间做一取舍。至少，具备一种优点，要比哪种都没有更强。

效率

衡量程序的真正效率，并不像乍看起来那么简单。首先，效率并不总能用计算机的运行时间来衡量——因为经常需要通过牺牲计算机的运行时间，来换取程序运行前后需要的时间。例如，在衡量编译器的运行速度时，一种普遍采用的标准就是统计其每分钟处理的卡片数目。如果我们根据这个标准，对一个汇编程序和（比如说）一个 FORTRAN 编译器进行比较，那么汇编程序很可能会显得效率更高——因为它的每张输入卡片中包含的信息量更小。有时也会根据另一种标准来衡量编译器的效率：每分钟生成的机器代码数目；但是这种方法也会带来另一个问题：一个在生成目标代码时更加“啰嗦”的编译器，很可能反而会得分更高。

无论如何，每个编译器都当然都是一个特殊的家伙——因为我们既需要考虑其生成代码的效率，同时也要兼顾到编译的过程本身。尽管编译器的效率问题无法用三言两语来说明，我们还是值得花些时间继续利用这个例子来说明：在关于效率的问题上，我们是多么容易被误导。比如，考虑根据“每分钟处理的卡片数”来比较对同一语言的两个编译器。这里的“同一语言”，就是第一个陷阱。实际上两个编译器支持的语言完全相同的情形极为罕见，因此，如果二者的确存在差别，我们就不应该简单地根据

“每分钟处理卡片的张数”这一指标来评判其效率，而应该同时考虑这两种语言的之间的差别。

就编译器的效率而言，源语言上的细微差别所导致的后果，可能会是惊人的。如果在编写编译器时忽略源语言的10%，而不加以实现，那么编译器的速度一般会提高50%。不幸的是，这10%的部分如何确定，却会因计算机以及编译技术的不同而差别很大——因此，语言的设计者将无法简单地为所有的实现环境，提供同一个“更加高效的”语言子集。

如果对技术规范其他方面的要求做类似简化，也可能带来效率上的类似改进。确实，在特定应用中，如果我们首要关心的是效率，那么第一步工作就应该是检查一下，看看在哪些方面的规范改变以后，可以提高计算机的效率——在这个阶段，不一定要顾及用户是否方便。当然，对通常的项目而言，在计算机上的运行时间并不是衡量代价的惟一标准。人工的时间同样需要花钱。诚然，我们可以把一些交叉结算的工作交给人来汇总，但是通过这种方式来节省计算机运行时间，其前提是，人用笔和纸进行汇总需要花费更多的时间。虽然我们可以很理性地做出这种选择，但是更经常出现的情况是，管理层并未对此给予任何指示，这种选择就做出了。

然而，对外部的功能要求进行裁减，却可能会与削减计算机运行时间的目的背道而驰。例如，要是为了简化计算机的运行，而把大量的、繁重的输入处理推给人的手工，那么我们将发现，因单次运行效率的提高而节省的费用总和，实际上远远比不上我们为多次运行所付出的代价。即使两个编译器处理的语言完全一致，只是生成的诊断信息和执行过程的监视信息不同，在比较其性能时，上面讲的这种效应也非常普遍。如果准备通过消除或削减错误检查工作，来节省20%的毛编译时间，那么我们最好首先确认：为了获得同样数量的这类信息，用户自己不至于可能要多花费40%的时间。

如果说，凭借简单方法去衡量单机、单任务环境的运行效率，还有可能碰运气奏效，那么与处理多机、多道程序环境中的效率测量相比，以其困难程序而论，前者只是小巫见大巫。比如，占用40KB内存空间、运行1小时，或是占用80KB内存空间、运行30分钟，哪种方案更好呢？这个问题的答案，取决于程序运行时，计算机的内部状态——姑且不论，同一

工作的每次运行，其状态都可能不尽相同。比如，某一天某一台计算机在1秒的时间内，在不影响到其他程序的运行性能的前提下，可能会允许增加40KB的任务负载；而等到第二天，同样是这个40KB的任务，却有可能会阻塞住另一个任务，在整整1个小时之内使后者无法运行。

正如我们现在经常看到的，在具有负载动态调整功能的系统中，多道程序运行的一些不稳定性会有所缓解。也正如程序开发的进度计划一样，用户所希望的并非程序的平均执行时间最小化，而是其标准偏差的最小化，这样他们才能合理地安排其工作进度。如果情况果真如此，我们就必须考虑从一开始到最后的整个工作进度，而不仅仅是“在计算机上的运行时间”。

我们将来的计算环境可能会发展成这样：在这种环境中，一个被称为“最好”的程序，应该无论内存容量大小、无论计算机配置如何，都能够顺利运行。当然，这种程序可能在任何一种配置下都表现平平——回忆一下Fisher基本定理，只不过现在是反过来应用它——但是其运行速度却能始终保持在一个稳定的水平上，因为它根本不必为了等待最佳的时间段而等上一整天。支持虚拟内存的计算机出现之后，我们可以无需专门的程序开发工作，就享受到这种灵活便利——尽管有迹象显示，只要对其特殊性加以考虑和利用，这种环境中的性能仍可以提高。

譬如，如果知道系统内存页面的大小，我们就可以对我们的程序进行适当的裁剪，使每段的大小正好与内存的一页吻合。这样，就可以避免过多的页面交换；此外，除了程序中的逻辑中断外，也无需跨越页面边界。然而，一旦我们这样做了，这个程序对内存页面大小不同的计算机的适应力就多少会有所降低。类似地，我们可以根据系统通常允许的页面空间，选择一个大小正好的算法；但是这也会带来类似的问题——即使计算机能够提供额外的页面空间，这样一个程序也无法加以利用；而反过来，要是系统提供的页面空间更小，那么这个程序的性能将会达到最差。

所有这些都告诉我们，运行效率正在成为计算中一个日益不明朗的问题。而且，随着单位计算能力成本的逐年下降，以及单位程序开发成本的不断提高，在程序开发方面比在产品方面投资更多的典型做法，早已不合时宜。如果那些被错划到“产品”名下的工作重新回到他们自己正确的归

属——“调试”——名下的话，这种不平衡就更加惊人了。但是无论采用何种标准，这种不平衡情况都将始终存在，而且还有愈演愈烈之势。因此我们期望，随着岁月的推移，我们能够听到的关于效率的言论将会越来越少——而与此同时，却越来越多地强调有效性。

小结

作为一个好程序，必须具备哪些要素呢？这个问题并不容易回答，甚至，可能它连一个正确的问题也称不上。我们必须考察到不同程序各自的优点，同时还要兼顾到其所在的环境。其中一些重要的因素有：

1. 该程序是否符合功能要求？或者更确切地说，它在多大程度上满足功能要求？

2. 该程序的开发是否按照计划完成？根据一些特定的方法，我们能够预测出的项目计划可能的偏差幅度会有多大？

3. 当条件改变时，该程序是否可能修改？修改的成本有多大？

4. 程序的效率如何？这里的效率是指什么？为了补偿某一个方面的低效率，我们是否会牺牲另一方面的高效率？

今后，尤其是在本书的后续讨论中，我们将尽量避免使用“优秀程序”、“优秀程序员”等概念——无论你认为这种概念是已经被广泛认同的，或者能够被广泛认同的，还是应该被广泛认同的。

思考题

请主管回答：

1. 你根据什么标准来确定应该奖励哪个程序员？在你的标准中，是否存在互相矛盾的现象（比如，一方面强调程序的效率，同时又要求程

序具有通用性)？你会很明确地告诉程序员，自己希望在他们的程序中找到什么吗？或者说，你是否只是告诉他们，你希望程序很快、很小、整洁、易于修改、错误很少，并且能在一周内完成？

2. 在你的计算中心里，各程序的适应性是如何积累的？修改程序是否属于你的一项主要开支？如果是这样，从你自己的程序开发实践出发，在你最初设定的目标中，有哪些可能导致这部分开支？

3. 在你的公司中，制定开发进度表的重要程序如何？你是否认为“失之毫厘，谬以千里”？或者你是否认为，在奖励程序员时应该更加看重稳定性，而不是偶然一次的歪打正着？即使一个计划不可靠，只要它是完成进度的惟一希望（尽管可能根本无法完成该程序），程序员还是会采用它——你知道这是为什么吗？

请程序员回答：

1. 在开始进行某个项目的时候，你的脑子里是否已经有了一些明确的准则？在确立这些准则时，你根据的是不是自己对于重要性的认识？或者，是依照上级主管的看法？在项目的进行过程中，这些准则有无更改？或者，你是否有某种办法，来确保其不被动摇？

2. 在编写程序时，你曾经有多少次想到过它可能在未来可能被别人修改？反过来，在修改别人的程序时，你又曾经咒骂过几回？

3. 你是否曾经因为追求“效率”而延误了工作的进度？反过来，是否曾经因为要“赶时间完成”，而没有做到尽善尽美？

参考资料

- Fisher, Ronald A., *The Genetical Theory of Natural Selection*, New York, Dover Publications, 1958.

请特别注意第 2 章关于自然选择的基本定理的介绍。

- Weinberg, G. M., *PL/I Programming, A Manual of Style*, New York, McGraw-Hill, 1970.

 请特别注意第 4 章关于在“优秀”程序的各种标准之间进行折衷的问题。

第 2 章　优秀程序的要素

评　　注

关于判断代码的质量，最重要的新因素就是经济的因素。一个畅销的而且能够带来收益的程序，显然要比不能畅销和带来收益的程序更好。至少有好几个不是程序员的人都告诉我：比尔·盖茨是世界上最伟大的程序员。在作为程序员的职业生涯中，我所编写的大部分关于操作系统的代码都没有被采用了，所以我很自然地怀疑这个经济上的标准。但是开诚布公地讲，如果编写程序的目的是挣钱，那么凭借自己编写的程序而挣钱最多的人，当然就是最优秀的程序员——从满足需求的角度讲，是最好的。

在原文中关于效率与有效性之间关系的故事里，读入卡片的工作显然已经过时了，但是无论使用的是何种输入介质，问题的实质并没有改变。在这些年里，我已经在这个例子的基础上进行了推广，终于发现了关于质量的定义：所谓质量，就是对于某个（些）人的价值[①]。对于那个年代的程序员，与功能的一致相比，运行速度是决定价值更重要的标准。有关价值取向的心理学研究成果，解释了为什么软件开发人员做出各自不同的选择——在以前，这一直是个难解的谜。

在软件质量管理的过程中，软件性能的偏差是极其重要的一个方面——在刚刚过去的几年中，这一点得到了确认。在平均性能方面，我们可以做得更好了；但是性能的偏差却仍然存在，甚至有所加强；在后

① 关于作者的这一观点，请参阅其后续著作之一：Gerald M. Weinberg, *Quality Software Management, Vol. 1: Systems Thinking*, New York: Dorset House Publishing, 1992。——译者注

一方面，用户仍然不甚满意。

20世纪70年代的男性至上主义者们会不经意地使用其特有的语言，很遗憾在我的原书中也出现了类似的词语，比如“美丽而谦恭的妻子”，以及“他”和“他的”——当时要是使用“她”和“她的”也许会更好。早在20年前，我就已经开始为此感到很窘迫。因为对这类写法所引起的对我个人的推测，我已经有了充分认识，所以在我后来的写作中，一直努力避免使用这类词语。今天重读这段原文，回味这种语言上的变化，我感到这是自己走向成熟的一个有趣的标志，但愿这也是我们的社会走向成熟的标志。25年前，我们中很少会有人能意识到，这种男性至上主义的写作风格会令读者如何猜测我们个人的品行。

许多主管似乎依然希望得到所有东西，却不知道更重要的应该是通过明智的权衡，得到自己可能得到的最佳成果。看起来似乎他们都认为，为了管理好纽约扬基队[①]，就必须要求每个击球手都能打出本垒打，同时要求每个投球手都能把击球手打发出局。

不出预料，在计算机运行时间与人工时间之间的折衷，今天的情况已经有所变化。

> “因此我们期望，随着岁月的推移，我们能够听到的关于效率的言论将会越来越少——而与此同时，却越来越多地强调有效性。”(p. 30)。

尽管如此，效率在一些（但不是所有）工作中依然很重要。我们还在研究这个问题。

本章末尾列举的问题，经受住了时间的考验，我对此实在感到惊讶，同时也不禁暗地窃喜得意。虽然所列出的参考书目并不属于计算机专业，但其情况也是如此。如果说人类在技术方面没有什么一致性，那么在人类的行为方面却不然。

① New York Yankees，美国著名的职业棒球大联盟队，颇具传奇色彩。1901年成立于巴尔的摩（Baltomore），队名金莺（Orioles），加入美国联盟；1903年迁至纽约New York，更名为高地人（Highlanders），1913年更名为扬基（Yankees）。自1921年起共得过36次联盟冠军和25次总冠军（其中1998、1999和2000蝉联三届），被公认为20世纪最伟大的球队。——译者注

第3章 如何研究程序设计

在本书的前两章里，我们已经尝试过将程序开发看成一项以人为主体的行为来加以研究。我们指出，虽然程序开发是人类行为的一种形式——一种复杂的形式——但是却没有什么人从这个角度来对其进行过研究。也许，人们之所以没有能够这样来看待程序开发这一过程，这其中是否存在着某种必然的原因？或许，程序开发的确是一项异常复杂的行为，以至于无从把握，所以在很大程度上，这个过程必然笼罩在神秘之中。

人类的知识必然是不完全的。我们无法在事先就确定，哪些是能够认识的，哪些是注定无法认识的。然而有一件事却是可以肯定的：如果不去进行尝试和努力，我们将永远不可能成功。有些读者对这种探索持悲观态度，对这部分读者，我只能建议他们，重新读读本篇一开始所引用爱因斯坦的名言：**“重要的是不要停止怀疑”**。

接下来的问题是，我们应该如何开始怀疑？既然程序开发是一项人的行为，似乎应该通过有关人类行为的科学来获取两方面的信息：一是那些可以直接应用于编程的研究结果；一是某些方法，这些方法可以帮助我们获得一些暂时还不能直接获得的信息。当然，这些方法也并非完全适用于对程序开发行为的研究——在这种情况下，正如社会科学家们成天在做的一样，我们必须创造新的方法。在此之前，让我们先看看，我们通常可以使用哪些方法。

自省

在人类行为学领域中，一些思维新潮的学生倾向于怀疑，自省这种方法能否称得上是科学的。尽管如此，自省却始终都是支撑这些领域的一块最基础的基石。此时此刻，我们也许会想起弗洛伊德（Sigmund Freud）的名言[①]：

> 学习心理分析首先要从自身入手，对自己的人格进行研究。“自省”一词并不能描述这一过程的确切含义，只是由于找不到更好的词来描述这一行为过程，因此我们可以暂且使用它。只要对这种方法有一定的了解，我们就可以把形形色色的平常而众所周知的精神现象作为素材，来进行自我分析。

也许我们还会记起 William James[②]（美国心理学的奠基人）的名言[③]：

> 简而言之，人类对于自己的思维状态的了解，显然要远远超过对与之伴生的大脑状态的了解。如果不对言语的精神要素进行自省式分析，众多学说的建立将是绝对无法想像的，比如心理学领域最耀眼的宝石——失语症学说。

诚然，在今天来看，这些论点确实来自心理分析学和心理学的草创年代，但是当我们着手开始对人类行为的一项崭新研究时，对我们而言，这种古老恰恰是其价值所在。我们当然不会在自省方法这里裹足不前——如果当初 Freud 和 James 是这样做的，恐怕他们就不会像今天这样闻名于世了。每当开辟一个新的研究领域或者分支领域时，即使没有任何精心的预

① Sigmund Freud, *A General Introduction to Psychoanalysis*, by permission of Sigmund Freud Copyrights, Ltd. London England.

② William James，威廉·詹姆斯（1842—1910），美国心理学家、实用主义哲学家。他在心灵与宗教领域的研究极为特别，是成立于 1885 年的“美国心灵学研究会”的主要创立者；他终其一生都在探讨超个人的心理现象与超心理学，认为人的精神生活有不能以生物学概念加以解释的地方，可透过某些现象来领会某种“超越性价值”；他强调人有巨大的潜能尚待开发，人的意识只有很少一部分为人所利用。他曾参与类似禅坐的静坐活动，表示静坐是一种唤起深度意志力的方法，可以增加个人的活力与生命力，也做灵媒的实证研究。——译者注

③ William James, *Psychology*, Cleveland, The Living Library, 1948.

防措施或者基于经验的方法，我们都可以从自己的头脑中直接收集到大量的信息。在一开始的时候，我们所需要的只是引导前进大方向的思想，而并不是什么有关后续过程的详尽而且逻辑严密的计划。

自省可以使我们洞察到什么？这里将给出几个实例，但是关于当前这个问题，其中有一个例子可以告诉我们应该如何区分方法的使用和方法的滥用。若干年前，有一次我在讲授 PL/I 时进行课堂考察，发现有一位学生在遇到如图 3-1 所示的语句时，表现出异常的困难。当他的程序最终完成时，我请他坐下来，针对他在这条语句处遇到的困难，回忆他思维的过程。

```
ANGLES(I)=2*ATAND(SQRT((S-A(IND(I,1)))*(S-A(IND(I,2))))/
                  (S*(S-A(IND(I,3)))));
```

图 3-1　冗长的 PL/I 语句

这名学生列举出如下的问题：

1. 他很难进行括号的匹配，因为其中的括号确实太多；

2. 原来的 ANGLES（I）是一个结构，而不是数组，因而编译器报告说，发现了一个非法的下标表[①]。但是由于下标表多如牛毛，他无法确定错误的具体位置；

3. 即使到最后可以运行时，程序还存在下列进一步的问题：

a. 由于 IND 矩阵位于清单的另一页上，所以很难发现；

b. 虽然括号现在已经匹配好，但是有一对的位置却错了——这是最难发现的问题；

c. 最后一个困难是由于精度问题造成的，而这个问题的起因，包括对不同数据类型的随意定义、过长的表达式以及除法运算——如果采用带有两个参数的 ATAND，除法运算就根本没有必要。

在诸多关于计算本身的问题方面，这样一个简单的例子，却可以让我们获得很多的启示：语句的合理长短、数据结构的选取、程序中不同部分的编排组织、使用括号而不是其他手段进行分解、编译器及其执行过程中

① 在 PL/I 语言中，一种用括号括起来的一个或多个下标组成的表。其中，一个下标对应数组的一维，所有下标一起惟一地标识该数组的一个元素或一个剖面。——译者注

的诊断功能的设计，以及程序开发教与学的方法。离开上面提到的自省，我们就无法获得任何此类的感悟。尽管如此，每天都依然有成千上万的程序员会遭遇这些问题，而他们却只是一味地抱怨说，自己的程序中有“错误”。如果不进行自省，他们注定要有这些“错误”；等到最后，他们将完全成为这些“错误”的奴隶。

虽然我们应该从自省入手，但是千万不要在此裹足不前，否则我们就会误入魔法或宗教的歧途，而不能踏上科学的大道。仅仅根据上面的这样一个实例，我们还不足以得出下面这些更具一般性的规律：

1. 人脑无法处理五层以上的括号；

2. 编译器应该提供更明确的诊断信息；

3. PL/I 的精度控制规则过于复杂，难以应用。

就程序开发心理学而言，上述每个命题都有可能成为一条“定律”。尽管如此，仅仅凭借一个关于自省的例子，还远不足以作为支持其成为定律的证据。为了获得一条“定律”，我们必须对其原理进行研究，以便对其应用范围做一界定——因为，每条定律都会受到这种限制。确实，通常对这种限定的了解，较之对定律本身的了解更为重要；而只有在对大量的案例进行过调查分析之后，才有可能明确这些限定。因此，如果离开自省，调查就不可能获得任何成果；而离开了广泛的调查，自省的价值也很难在应用中令人信服。

观察

与自省顺理成章的下一步是观察，我们要观察人们到底在做些什么，而不是他们自认为在做什么。比如，如果要对括号的层数做进一步研究，我们就可以观察一下别人使用的层数以及他们在使用括号时所遇到的困难。但是，在观察时需要注意的一个问题是：观察只能告诉我们人们确实在做或做过的事，而不一定就是他们能做的全部。因此，即使在对几百名程序员进行观察后，没有发现任何人使用过超过五层的括号，我们也不能据此得出“结论”：没有人能够使用六层括号。人们可能会怀疑自己无法做到某些事，这类的观察确实可以平息此类猜测——因为，一旦我们找到



一名程序员，他能够使用至少六层括号，我们的问题就解决了。

也可以说，我们至少解决了部分问题。而在观察时需要注意的第二个问题就是：要搞清楚我们究竟要观察什么。一名程序员在任何情况下都绝对不使用第六层括号——这背后可能有某种原因，使得他无法成功地使用第六层括号。一旦观察到一例六层括号，我们的工作才只是刚刚开始；接下来，我们需要对促成或者妨碍这一案例发生的环境条件进行界定。这种界定能够进行到何种程度，取决于我们希望借助观察的程度。比如我们正在设计一种程序语言，并且为保证编译的效率起见，可能会将括号嵌套的深度限制为五层。这时我们就需要进行评估，看看因无法使用第六层括号而感觉不便的情况有多少。但是，我们却不一定非要知道这些人为什么要使用第六层括号。

但是正因为程序开发是一项极其复杂的行为，所以对于此类观察，我们需要格外谨慎。比如说，很多程序语言都将数组的维数上限设置为三——这没有任何来自心理学的依据，而只是鉴于人们不经意地观察到，似乎没有人会使用第四维。除偶然性外，这个观察的差强人意还反映在其他很多方面。首先，如果程序员使用的程序语言只支持三维数组，我们当然无法观察到使用第四维的例子。其次，即使支持超过三维的数组，由于受到该语言其他特性的限制，程序员仍有可能无法有效地加以利用——也许是因为无法设计一种格式，以支持打印输出更高维数组；也可能是因为下标定义得过于繁琐，超过三维以后显得很乱。这里的关键在于，当试图把在某个环境的观察结论带到新的环境时，我们应该多加谨慎，因为新、旧环境之间可能在很多方面存在差异。

在观察时需要注意的第三个问题，是观察者与被观察的被试[①]之间的干涉现象——这是另一种形式的测不准原理。社会科学对此问题关注已久，并且将其总结为“霍桑效应”（Hawthorne Effect）[②]；在这方面，我们可以从中获得很多有益的参考。该效应得名于西部电气公司（Western Electric Company）所属的霍桑工厂，1924～1927 年期间在此进行的一项

① Subject，被试。心理学中指参加并接受试验的人。——译者注

② 指因受到他人关注而带来的提高或进步。比如，工人、学生等因受到研究人员的关注而增加产量或提高成绩。——译者注

有关工业心理学的实验以失败告终。失败的原因是：在实验过程中，无论工作条件如何变化，生产效率始终一路攀升。实验者到最后才意识到，他们忽略了一个重要的因素——因为受到关注，在工人们内心油然而生的自豪感。就这样，他们所观察到的现象，实际上都是由观察着自己的观察行为本身产生的。

观察者与被试之间的这种干涉现象，不仅会发生在工业心理学领域，而且也必然存在于任何有关人类行为的科学领域。在其研究领域中，人类学家们始终都在为一种挥之不去的恐惧所困扰——他们总是担心，自己所观察到的情况，不过是为他们这些上门访谈的人类学家而刻意编造出来的。这些人类学家试图通过“参与式观察”(Paticipant Observation)[①]，使得自己成为被观察对象眼中的“隐形人”。他们采用的办法，就是融入到被观察的文化氛围之中而不会被察觉；与此同时，该文化得以继续运转，仿佛没有任何外人在场一样。显而易见，参与式观察的方法当然可以应用于对程序开发的研究；我们的学生通常是正在工作的程序员，本书中报告的一些数据，正是他们通过充分利用这种方式而获得的。

在观察过程中做到“隐身”的另一种方法，就是让被试根本无法察觉到正在被观察。这类方法有很多，它们全部都可以归于一类所谓“不为人察觉的测量方法”。我们没有必要在此详尽地讨论此类方法，因为此方面的书籍与论文已是连篇累牍，其讨论问题的范围不仅包含方法本身，而且也涉及在使用它们时引发的道德伦理问题。但是，我们还是会讨论这样一个“不为人察觉的测量方法”；在有关计算机的研究方面，这种方法多少会显得有点与众不同——实际上，这种方法是通过计算机本身来记录用户的行为，因此用户难以察觉到它的存在。

在某种程度上，任何一台进行自我记账的计算机在这样做的同时，也在对程序开发的行为进行不为人察觉的观察。针对分时系统中用户在终端上的行为，已经进行了一些明确的研究（有关此项工作的综述，请参阅：

① 相对于非参与式观察（Non-paticipant Observation）。研究人员以观察参与者（observer-as-participant）的身份“混迹”于被试之间，从而将观察过程中的干涉效应降至最低限度。在田野工作中，这种形式被广泛用于访谈调查。——译者注

Sackman[①]，1970，chap. 2)。一般而言，这些研究都是基于统计的，但是由于计算机对数据的记录非常详实，所以即使是经过很长时间之后，这些记录也仍然可以用以对某个人进行研究。实际上，也可以说正是因为这些记录过于详尽，所以才迫使这方面的研究不得不以统计的方式进行——从这个意义上讲，如果希望成为一名观察者，那么你面临的最重要的问题就是：应该丢弃哪些数据？一个人坐在终端前每个小时内生成的数据，要远远超过一名心理学家在一个小时内能够确有成效地分析出来的结果；即使是 50 人共享同一系统，情况也不会有任何改善。

然而我们不能被这异常巨大的数量所迷惑，不要据此就认为，这些规模庞大的数据里必定含有我们所需要的信息。尽管系统平日里生成的记账和日志信息非常便于使用，但是在任何一次认真的研究中，通常都需要借助一些硬件或软件来找出我们所真正寻找的东西。比如，计时信息对于记账很有必要，但是对于心理学研究而言，还远远不够。首先，计算机使用的计时单位与我们要研究的不同，而且很少有系统会保留每次事务处理的日志。其次，用于记账的计时其分辨率通常为一秒甚至更长，但是在许多心理学研究中，我们需要精细到毫秒的级别。

实验

对通过观察所得到的庞大数据进行处理需要付出高昂的代价，为了降低处理的代价，一个办法就是设计一些实验。通过进行实验，数据量得以减少，同时其中与我们感兴趣的行为有关的信息却增加了。但是实验中会有很多危险，比如第一个危险恰恰就是：由于实验经过了高度的精炼，以至于可能遗漏了我们最感兴趣的数据。另外，实验的特殊条件也会限制被试的行为，以至于我们再也无法观察到在自然条件下所观察到的结果。

这类限制对被试行为的影响有的属于计划之中，也有的会出乎预料。作为一个例子，Sackman 引用了多项对分时和批处理系统进行比较的研

① 参见本章末尾的参考书目。

究，其中的批处理系统是通过模拟实现的：以分时方式运行任务，然后故意把结果保留一段固定的时间，再发送给用户。这种模拟的用意，本来是为了保证两组程序员使用同一种程序语言，并且拥有完全一致的系统设施。但是和真实的批处理系统相比，以这种方式模拟实现的批处理系统不尽相同。比如，根据终端用户的需求，对使用的语言进行了裁剪。一些面向终端的系统命令无法使用，但却不能通过面向批处理的命令代替。面向终端用途的诊断信息成为多余，而由于终端用户更容易做自行调整，批处理系统本应提供的调整措施现在却可能没有。最后，过于规则的运行时间，也可能产生一种更为微妙的影响。正常的批处理系统接受提交的任务后，绝不会在严格的 2 个小时后给出计算结果；实际上，运行时间的这种波动，正是影响用户行为的主要因素之一。由于不经意地消除了这种波动，实验的数据注定会在这个或那个方向出现偏差。为确定并校正这个偏差，必须进行另一项研究。

另一项研究！是的，问题难就难在这里！任何研究都需要经费，与对人的行为的其他方面进行的研究相比，对程序开发过程进行的研究需要的经费更多。比如，在一项小型研究中，需要在 3 个月内占用经验丰富的 9 名程序员 1/4 的工作时间。他们的平均年薪是14 000美元，再加上适当的管理费用，这 9 位程序员一年的开销就算20 000美元吧——换言之，在整个研究期间，需要为每位程序员支付1 250[①]美元的实验酬金。再加上实验人员本身的工资，机时以及名目繁多的后勤和秘书服务费，这类实验的预算动辄超过30 000美元——而这仅仅是雇佣 9 名被试时的数字。

在传统的心理学研究中，需要的费用没有这样高。首先，尽管时髦的做法是更多地使用计算机，但是传统研究并不需要计算机。而更重要的是，那时被雇佣的被试通常是免费的。对此问题，有人曾经这样调侃道："心理学是 18 岁的大学新生的心理学。"此人的意思是说，被试通常来自大学中新生的心理学课，这些课要求选课的学生至少参加一项实验并在其中作为被试，这是获得相应学分的必要条件之一。这些被试通常不会获得任何报酬，更不用指望能够像程序员一样，每年能拿到14 000美

① $20000×（1/4）工作量×3 月/ 12 月＝ $1250。——译者注

元了。

在选取被试时，可能会夹带进不经意的限制条件。尽管上面那种所谓的心理学还勉强可以称做是大学新生的心理学，但是关于程序开发的心理学却很容易蜕化为实习程序员的心理学——同样地，其原因也是在于费用问题。相对于正式的程序员，实习生的报酬更低；然而更重要的是，经验丰富的程序员从事的工作往往是至关重要的，而实习生们则不是。因此，尽管经验丰富的程序员很适于参加实验，但是往往很难请到他们。后面，我们将围绕专业与业余程序开发之间的差异进行讨论——那时我们将看到，在我们对程序开发进行思考时，这种混淆现象是多么普遍，并且如何把我们蒙蔽起来。如果在对编程过程的心理学研究中，因为费用和方便的考虑而坚持使用实习生，那么我们只会将由此得出的一些似是而非的结论误作为“科学的证据”。

毫无疑问，某些实验的确可以通过雇用实习生很好地完成——然而在这种情况下，实验人员必须负责地进行验证。而更常见的情况是，根据实习生们的数据所得出的结论，只限于与实习生们有关的应用——也就是说，只会对培训教程的设计有帮助。与今天的情况一样，以前曾经有很多人对 OS/360 的任务控制语言（Job Control Language—JCL）颇有微词，我们曾经为此进行过一项试验，希望了解并确定该语言在使用方面的困难程度。在对 OS/360 课上的新手进行测试之后，我们发现困难确实存在；但是得益于精心设计的练习题，随着学习的深入，这类困难或多或少都会逐渐有所降低，到课程结束时则会完全消失。如果只对初学者进行测试，然后就仓促做出结论，那么我们的确就会认定，OS/JCL 很难使用；而正确的结论应该是：该语言只是很难学习。无论是通过课程还是令人痛苦的练习，一旦掌握了这种语言后，你将会发现 JCL 其实并不比其他的任何语法严格的程序语言更难用。尽管该语言本身或许需要重新设计，但是通过改善培训方法，也可能会更富有成效。反过来，如果只对程序开发老手们进行测试，那么我们就可能会认为，所有关于 JCL 的抱怨不过是大惊小怪。只有在对两类人都做过测试——或者按照我们在实际中所采用的方法那样，从实习生阶段就开始对程序员进行追踪调查，直到他们成为训练有素的熟练工——之后，我们才能够得出关于

JCL 的正确评价。

然而，无论是雇佣程序开发老手还是实习生，我们都会碰到另一个问题——难以对程序员进行这种简单的分类。在对 JCL 进行的研究中，部分学生在程序开发方面的确是新手，但是另一些人却很有经验。当然，就 JCL 而言他们都是新手。在这种情况下，如果简单地把测试结果与“从事程序开发实践的年头”关联起来，并将其作为衡量“程序开发经验”的尺度，那么我们根本就得不出任何结论。在课程起初进行调查时，有经验的和无经验的学生的成绩同样糟糕；而到课程结束时，这两类学生的成绩又同样优异。因此，在这个问题上，是否有“经验”一点关系也没有，重要的反而是关于 JCL 的、特定的经验。

Sackman 曾经对“有经验的”程序员进行过一项研究，这项研究的结果说明，要判定程序员经验的有或无，的确不是一件轻而易举的事情。在这项实验中，除了其他标准之外，还通过“从事程序开发的年头”这一指标来衡量程序员经验的多少，其范围从 2～11 年。在其中一个问题上，性能最好和最差的两个程序，分别出自于经验值同为 11 年的两位程序员之手。显然，除了单纯的“从事程序开发的年头”长短之外，必定还有其他因素在起作用。一种可能的解释是：程序开发的经验所覆盖的方面过于分散，以至于仅仅根据工作年头长短这一项指标，无法成为有效的评判尺度；另一种解释是：即使是经验值（参与程序开发年头数）相同的程序员，他们在此期间学习到的具体知识内容也可能不尽相同。

在将关于经验与测试的讨论告一段落之前，我们还需要对另一个重要的问题进行研究。这个问题之所以显得重要，是由于这样一个错误观点的存在：程序开发是一项个体的行为，而不是社会性的行为（这个观点在此前很少受到质疑，但是在本书中，我们将要花费大量的笔墨来对此加以推敲）。如果实验人员轻信这个观点，他们就必然会去选招一个个孤立的程序员（无论是有经验的，还是经验不足的），而且不会认为把他们作为观察对象有什么不对劲的。但是正如我们现在已经看到的，有充分的证据可以说明，从程序开发社会集体的层次上对程序开发过程进行研究，才是正确的研究之道。程序员们与他人协同工作（而不是单兵作战）的时间，平均要占其全部工作时间的 2/3——是的，这是一个事实。在个人层面上的

研究并非无足轻重，但是如果考虑到上述的事实，我们自然就会提出质疑：为什么99%的研究都是针对程序员个体进行的呢？

当然，有人可能会辩解道：如果说研究程序员个体的成本昂贵的话，那么研究程序员**群体**简直就是铺张奢侈了。此外，并非任何程序员的“群体”都可以用来研究——确实如此。为解决某个问题而临时拼凑起来的一群人（比如，由若干实习生拼凑起来的“团队”），并不能称为“团队”——看看所有的全明星队，那种松松垮垮、一盘散沙的表现，你就可以很容易理解这一点。有一个神话说：“程序开发是人之个体性的最后堡垒”，今天组建的团队就成长在充斥着此类神话的环境之中——正因为如此，只研究这些团队也是不够的。

个体式的研究方式，必然会强行将单个个体从其正常的工作环境中剥离出来。这种漠视社会效应的做法，必将引起人们对它的疑问。在我们的一项研究中，有一名程序员在写完了程序之后找到我，询问应该由谁来对他的程序进行测试。在他所属的团队中，这是开发过程中的一道标准工序；但是在当时的研究过程中，这却是被禁止的——否则，就会使实验结果“失效”。如果在研究青蛙游泳的行为时，有人将它们的腿割下来放到水中，然后试图观察到腿会游泳，那么一定会让人“贻笑”大方的。然而我们在笑过之余回头想想自己，那种强迫程序员在工作时与外界隔绝的做法，与割青蛙腿的做法又有什么两样呢？然而事实上，的确有很多人都习惯于这样一种推理的逻辑：既然青蛙是凭借腿来游泳的，那么为什么还要去研究青蛙的其他部分，把一件本来“显而易见的”事情搞得如此“复杂”呢？

心理学测量

程序开发心理学家在测量方面遇到了很多问题，上面提到的用“程序开发实践的年头”来衡量“程序开发经验”的问题，只是其中一例。在有关人类行为的观察和实验中，可以测量的指标很多——如果说数以百万计有点夸张的话，那么随便输出几千个却是轻而易举的。我们可能需要记录年龄以及敏捷程度、眨眼频率以及出生年月序列、颅容积以及习惯去的教

堂、演绎推理能力以及获得的学位、心电图以及初等数学、蹙眉的频率以及父亲的职业、习惯打的手势以及年级平均成绩、身高以及憎恶的东西、想法以及理想、是否神经过敏以及是否爱说行话、慈爱程度以及知识水平、是否左撇子以及是否有潜在的同性恋倾向、母亲是否有洁癖以及肌肉的紧张程度、是否操当地口音以及是否有神经性抽搐、所持的观点以及接触到的事物、偏爱的食品以及瞳孔的移动、被问及的问题以及回答过的问题、回答问题时的反应速度以及对帮助的要求、支持的来源以及发音错误、每天的工作时段以及是否用脚敲地板、在大学所获的学位以及不合常规的举止、词汇量以及投票记录、体重以及工资、X 光照片以及仇视外国人的指数、打哈欠的方式以及在大学念书的年头、接合性以及生辰八字[①]——毫无疑问，在所有的科学家之中，心理学家是最爱管闲事的。

然而这还仅仅只是问题。在所有科学之中，仿佛心理学可能测量的事物是最多的——多得令最通情达理的人也难以理解。那么，是否这种冗余现象对心理学而言是独一无二的呢？并非如此，实际上除这方面外，心理学与其他行为科学一样，都处在成长的幼年时代。伟大的物理学家麦克斯韦（Maxwell）曾经有句名言："认知，就是测量"（To Measure is to Know）——这是很多科学家的座右铭。虽然麦克斯韦的意思可能是说"认知就是知道如何去测量"，但是如果理解为"认知就是知道要测量什么"，也许会更好些。因为经过长期的经验积累，物理学家们已经懂得，什么是值得测量的、什么又是不必测量的——因此在物理学领域中，可能需要的测量看起来很少。由于懂得什么才是值得测量的，物理学家们才能把他们需要进行实验的范围集中到这些关键位置。就某种意义而言，物理学就是关于物理学家们所能够测量的事物的科学。

但是在研究人类的行为时，我们却还不能享受这种简单性带来的便利。我们从事的学科，尚未被定义为一门"关于什么是我们能够测量的事

① 作者在此一口气列举了 26 对指标，其字头依次对应 26 个字母。通过这种修辞形式，作者希望告诉读者：心理学测量用到的指标不仅多如牛毛，而且虽然从表面看它们之间似乎有某种联系，但是这种联系都是牵强附会、似是而非的——就像它们的首字母可能相同一样，完全是一种巧合。——译者注

物的”科学，而是被定义为“什么是我们希望了解的”——亦即人类的行为。因此，我们还不得不在林林总总的、可以测量的指标中搜索，期望着通过其中一个或少数几个量，就能够获得一些领悟，然后再尝试根据它们来回答我们最初的问题。在这个搜索的过程中，任何蛛丝马迹都不能放过；无论是什么样的观点，如果你认为它**先验地**就是荒谬可笑的，那么最终可笑的必定就是你自己。比如刚才提到的生辰八字，它究竟是不是一项不合理的指标呢？你又是如何做出这个判定的呢？我们并不赞成占星术，但是至少医学家已经懂得，对于来自民间的秘方和偏方，最好还是尊重一些更为妥当；同样地，在进行人类行为的研究时，也需要这样的谨慎态度。

事实上，最近有几项研究结果就声称，出生月份或季节，与一些心理的甚至身体的指标有关。这些结果是否能够经受时间的考验，是否不过是骗人的把戏，与我们今天讨论的问题已经没有多大关系。我们应该从这些问题中看出：如果不能摒弃所有先入为主的偏见，那么必然会由于轻率地将某一因素拒之门外而犯错——作为严肃认真的科学工作者，这才是对我们来说至关重要的。鉴于已经在黑暗之中摸索了这么久，所以我们当然舍不得错过“民间方术”和被试，即使这些很可能最终被证明是毫无根据的，甚至是彻头彻尾谬误的。

来自民间的方术，不过是通往顿悟的一个途径。另外的一个途径，是目前正在服务于其他一些场合的工具。因为这类工具确实存在而且有效，另外，我们也至少对其有了一定了解，所以应该允许利用这个工具，让它们在我们所进行的研究中发挥作用。因此，既然我们已经有了“智力”测试的方法，那么我们首先需要进行的一项工作，就是对程序员进行这类测试，以期在测量结果（“智力水平”）与我们希望理解的对象（“优秀的程序”）之间，发现某种联系。如果运气好的话，我们确实可以找到一些联系。

而最常规的途径，却是走另一条路——根据待解决的问题来确定要使用的工具。Sackman 的工作即是这样的一个例子——他希望了解分时系统的效能，因此他量身定制了几项研究课题，并对大量的指标进行了测量。另一个例子，是 ACM 人事研究专用业务组（ACM Special Interest Group

on Personnel Research)。从管理（特别是人事管理）的角度来看，这种松散的组织形式主要对程序开发感兴趣。因此，他们的研究课题，往往倾向于人事选拔与培训方面，而且他们所作的测试只反映这些兴趣点。就我们自己而言，很多兴趣点集中在对程序语言的设计上，在确定要做哪些测量时，这方面兴趣对我们的影响与其他因素不相上下。

然而，如果对自己将要研究的对象知之甚少，那么你就不可能在事先知道真正需要测量什么。因此，你在开始阶段所进行的实验，和你的最终目的并不吻合——这些初期实验的作用，不是要得到最后的测量数据，而是要确定哪些是可以测量的以及哪些是值得测量的。二者之间的差异，类似于人类学和社会学中的方法学家们之间的差异。社会学家工作和生活在同一个文化氛围之中，他们有理由假设自己对该环境有充分的了解，所以可以设计出一张满足测量要求的调查问卷。但是人类学家们的情况则不同，他们无法使用这种调查问卷，因为他们无法确定，在另一个不同的文化氛围中，哪些问题才是有意义的。所以概括起来，我们可以说，如果说社会学家们所追求的是答案的话，那么人类学家寻找的则是问题本身。

就目前而言，在程序开发心理学方面将要进行的大多数工作，还不得不限于先去“寻找问题本身”。尽管我们可以利用心理学中现成的标准测量工具——比如调查、测试等。但是实际上，与其说我们在研究被观测对象，还不如说我们在研究测量的方法本身。根据这个观点，我们最好不要把这些似是而非的测量数据，打上诸如“测试分数与工作经验之间的相关度为 0.72”的幌子，然后招摇过市。尽管我们可以想当然地根据主观愿望，把“测试分数”精确定义到小数点后的任意位置，但是我们却永远也无法把“程序开发经验”测量到小数点后两位，甚至连小数点后一位也做不到。所以，0.72 中的数字“2”完全是统计中的噪音——而且同样地，数字“7”也可能是如此。

限于篇幅，关于心理测量学或心理统计学的整个领域，我们就不一一赘述，读者可以参考后面几篇精辟的参考文献。行为科学家们已经碰到过太多的陷阱，在没有对这些陷阱进行研究之前，我们不能企图根据在仓促间得到的这些数据，得出任何确切的结论。在后续讨论中，我们还将有机



会根据特定的实例，进一步对此加以说明。

利用行为科学中的数据

正如我们讨论过的，在行为科学中以前所积累的经验能够为我所用之处，恐怕主要是在方法论方面。鉴于程序开发过程的与众不同，不能指望其他领域中现成的结论，可以直接被移植到我们的研究中来。通过这些结论，我们只能获得一定的领悟，但是绝不能指望马上就得到最后的答案。

我们到哪里才最有希望获得这种领悟呢？如何获得呢？最好的策略，是不要沉迷于那些过于具体的理论而无法自拔——因为我们最终总会发现，这些理论所依据的前提条件，与程序开发过程的实际情况相去甚远。比如，虽然有关“问题求解”的心理学研究成果多如牛毛，但是这些成果并不适用于对程序开发的研究，因为在这个领域里，“待解决的”那些“问题”简直太简单了。实际上，这些问题不可能不简单——因为该领域中几乎所有做过的实验，都是基于这样一个观念：任何“问题”都有一个正确“答案”，而且心理学家知道这个答案。

然而对于程序开发而言，我们可以肯定地说，没有“正确的”答案。退一步讲，即使存在这样的答案，我们也可以肯定，我们考察的对象们要比我们更加了解它。刚上路的学生所得出的“解答”，在某些方面往往要比老师或程序开发老手们更为出色——这种事情是屡见不鲜的。因此，从各种问题求解方面的文献中，我们不可能直接得到什么结果；但是毫无疑问，我们将可以从中得到某些领悟。

我们可以从如此众多的领域中获得有益的启发，以至于我们不得不对自己的陈述进行一定的整理编排。经过这一整理过程，可以帮助我们牢记那些重要的思想和建议；与此同时，通过结构中存在的缺陷，将有助于我们确定可以向哪些领域寻求帮助。但是，我们千万不要把这种经过整理得到的结果太当一回事——因为，这只不过是一个提供方便的工具而已，绝不是镂刻在由某个人从某个高山之巅背下来的某几块石板之上的一整套现

成的真理[①]。

在社会科学之中，为我们针对计算机程序开发的研究提供最有用的整体模型的，是人类学。只要有一点艺术性的豪迈和想像力的拓展，我们就可以认为计算机程序员们拥有自己的某种文化——一套他们共同拥有的信念和行为——而他们日常的举止，正是由这种文化决定的。在进行有关程序开发的研究时，我们必须首先考察一下这种文化的“社会结构”——也就是程序员之间、程序员与其他非程序员之间互相沟通的方式。我们将发现，通过一些令人吃惊的方法，可能打破和改进这一领域中当前的常规——与我们第二个方面（从个体行为的角度对程序开发进行）的研究相比，在这方面获得的进展将更为显著。尽管只有在延伸我们的想像力之后才能进行这种类比，但是这方面的研究的确与人类学中有关人格和文化的研究关系密切。我们将看到，每个人会如何将其个性灌注于自己编写的程序之中，而程序开发的行为反过来又会如何对程序员（的个性）产生影响。在关于程序开发的社会结构的讨论中，我们将大量借鉴社会心理学；但是在这里，我们将首先借鉴个体心理学——尤其是在对问题求解以及人格等问题进行研究时。

这两个小节将为后续的深入讨论做好铺垫，并为我们提供一种正确的心态——只有持这种心态，我们才可以从心理学的角度，来考察程序员使用的各种工具——程序语言、操作系统以及其他设备。这些工具是程序开发曾经创造的“物质文化”的遗迹——为了理解这种早已消亡的程序开发文明，程序开发的考古学家们将发掘出这些古老的石器，并对其加以研究。与此同时，程序员们并非生来就处在这个文化氛围之中，要进入这个圈子，他们需要一些机会、经过一个过程。因此，我们还需要研究他们会如何（或者说，应该如何）溶于这种文化之中，以及他们会如何习惯于这种复杂的生活之道。在这个问题上，关于学习——尤其是语言学习——的心理学将发挥重要的（虽然不是惟一的）作用。

最后，在成功地完成了所有这些工作后，我们将发现，我们心态会变

① 作者在这里暗指的是十戒（Ten Commandments 或 Decalogue）。据《旧约全书》（*The Old Testament*）记载，在西奈山（Mount Sinai）上，神用指头在两块石板上写下十项戒律，赐颁给摩西。——译者注

得足够的客观。在这个时候，我们就可以对有关程序开发的某些神话进行考察——这些神话被很多程序员奉为对程序开发的信条。正如伯特兰·罗素（Bertrand Russell）[①] 所言：所谓信仰，就是对那些毫无证据的事物的信任；也正如安布罗斯·比尔斯（Ambrose Bierce）[②] 曾经定义的一样：所谓神话，就是对（与自己别无二致的）其他人的顶礼膜拜。也许我们需要的只是坚持原有的信仰、绝不动摇；然而我还是希望通过自己的努力，能够戳穿一些仍不为人所知的神话——哪怕只有一个这种神话因此消亡，这本书也就不是白写的了。

小结

各种行为科学的研究已经得到了大量的方法和结果，因此在对程序开发这种内容丰富而且过程复杂的行为进行研究时，我们需要借鉴和利用这些现有的资源。然而这个借鉴的过程不可能一帆风顺、毫无风险，在为程序开发心理学的研究进行规划时，我们需要特别地谨慎提防，以免落入陷阱。其中的主要问题包括：

1. **尚未确定其有效性，就贸然进行自省；**
2. **在过分狭隘的基础上进行观察；**
3. **错误地选取观测的指标，正确的指标反而被忽略了；**
4. **与被观测环境发生相互干涉；**
5. **生成的数据太大，而其中的有效信息却不足；**
6. **对实验的限制太多；**
7. **选择观测对象时，过于依赖于实习生；**
8. **忽略对群体效应、群体行为的研究；**
9. **测量的指标过于简单；**

① Bertrand Russell（1872—1970）。英国哲学家、数学家、社会学家，也是20世纪西方最著名、影响最大的学者和社会活动家。1950年，获诺贝尔文学奖。——译者注

② Ambrose Bierce（1842—1914）。美国19世纪后半期的讽刺作家。在他的《魔鬼词典》（*Devil's Dictionary*）中，其愤世嫉俗的文风体现得淋漓尽致。——译者注

10. 精度未经核准，即引用有关统计数值；

11. 把其他领域的结果，应用于并不适用的环境。

在任何一个刚刚起步的领域，我们都可能会犯这样或那样的错误，然而或许在其中，最大的错误就是（因）过于谨小慎微（而裹足不前）。虽然我们需要尽可能地循规蹈矩，但是更重要的却是要敢于进行试验；只要敢于进行试验，哪怕结果是失败的，也要比什么都不做更好。

思考题

请主管回答：

1. 你所在的计算中心里，采用了哪种记账系统？你在对程序员的表现进行评估时，是否使用其提供的数据？这些数据是否是针对这方面用途的？为了使这些数据对你的评估更有帮助，你的记账系统应该如何改进？

2. 对于手下经验丰富的程序员，你是否会允许他们参加行为科学方面的实验？对于实习生呢？对于其中反对参加此类实验的人，你会采取何种措施？对于实验员，你会在安全措施方面有哪些要求？

3. 你是否曾经在以下期刊中，阅读到过有关心理学的文章：《哈佛商业评论》（*Harvard Business Review*）、《读者文摘》（*Reader's Digest*）、《花花公子》（*Playboy*）、《思考》（*Think*）或者《今日心理学》（*Psychology Today*）？其中所报告的一些结果，你是否曾经在自己的工作中加以应用？在把其他环境中得到的数据移植到对程序员的管理方面，你有什么经验？

请程序员回答：

1. 在发现一个程序错误后，你是否会先静下来，然后尝试着探寻一下导致错误的思考过程？在下次发现某个错误时，请尝试着这样做一次，然后根据你的体会写一个简要报告或概述。

2. 你是否有对工作表现进行记录的习惯？你对哪些方面进行记录？你的记录精确到什么程度？与简单地对自己的工作进行反思相比，通过这些观察，你可以得到哪些通过简单的反思无法获得的体会？根据你所在计算中心的记账程序所提供的信息，在关于自己的行为、同一计算中心内的其他程序员的行为方面，你是否得到过什么启发？

3. 关于参加对程序开发过程的心理学研究实验，你是否有什么反对意见？为了令你放弃这些反对意见，你认为应该为你提供哪些保证措施？如果你确曾参与过某项心理学实验，那么请就其中你印象最为深刻的某个方面，写一个小型报告或总结。你是否可以看出，就程序开发而言，为什么其中的部分结果是有效的，另一些又是无效的？是否曾经有人向你介绍过这类研究的结果？你是否理解他们所解释的原因？

参考资料

□ Freud, Sigmund, *A General Introduction to Psychoanalysis*, Garden City, N. Y., Doubleday, 1953.

除了其本身的及其自然而生的意义之外，这本书也是一份很有用的历史性文件，通过这本书，我们可以了解到任何一门人类行为学起步的一般方式。就目前正在讨论的话题而言，该书具有特殊的意义，因为透过这本书我们可以了解到，Freud 如何向其读者证明，他所讨论的问题确实存在。Freud 所使用的方法之一，就是对错误——诸如所谓的“口误”① 等——进行讨论。这种特别的讨论方式，对从事程序开发工作的人们具有特别意义。

□ James, William, *Psychology*, Cleveland, The Living Library, 1948.

作者因“首次将现代科学的方法及精神扩展到对人类思维的研究”而闻名，虽然这本经典之作中的部分内容已经过时，但是这些只是很小的部分。James 的工作与 Freud 一样，为计算机程序开发心理学的学生提供了丰富的研究成果；这两本书堪称该领域研究极佳的

① Slips of the tongue，又称“弗洛伊德失语”（Freudian Slips）。Freud 指出，人们在讲话时出现“舌头打滑”的现象，表面看来似乎是口误，其实不然——这只不过是在心理抑制松弛后，把本来想说但又（因为胆怯、顾忌等）不能说的话说了出来而已，是说话者内心的一种暴露，反映了深藏于其潜意识的信仰、思想或情感。弗洛伊德同时指出，人们也常常会故意“口误”（比如说某位总统如果来自某个特别乐于“指点”别人的国家，就很可能在“无意”之间将人家的某个省，说成是一个“国家”）。——译者注

启蒙教材，惟此才不致割断历史的发展脉络。

□ Brown, J. A. C., *The Social Psychology of Industry*, Baltimore, Penguin Books, 1954.

这是关于工作群体的研究实践的一份非常易读的说明，它使我们得以从历史的角度理解诸如霍桑效应等问题。刚刚涉足于工业心理学领域的那些人，将会对其中的研究结果留下深刻的印象，所以本书非常值得他们一读；尽管工业心理学的研究与诸如程序开发等复杂问题的研究没有多大的关联，但是对于与此领域密切相关的那些人来说，还是有必要读一读的。退一步说，至少对于我们可能做出的过分之举，Brown 关于工业心理学草创初期的综述，将会提醒我们加以避免。

□ Junker, Buford H., *Field Work——An Introduction to the Social Sciences*, Chicago, University of Chicago Press, 1960.

该书作者（似乎不无道理地）声称：只有亲自进行过田野工作[①]的人，才有资格对其作正确评价。但是在承认其认识的局限后，他竭尽全力地证伪了这个论断；而读者则得以从中感受到社会学和人类学中的各种田野工作方法。

□ Golde, Peggy, ed., *Women in the Field*, Chicago, Aldine, 1970.

书名中的“Women”一词带有一点歧义，其实该书中所揭示的一个结果就是：田野调查者之间由性别而引起的差异是多么的微乎其微。可以从几个不同的层面来阅读该书：既可以完全作为野史轶事来走马观花式地欣赏，也可以当成关于人类学田野调查方法的一份全面综述（虽然没有进行编排）。

□ Hammond, Phillip E., *Sociologists at Work*, New York, Basic Books, 1964.

与“*Women in the Field*”相同，这也是一本关于方法的轶事专集，然而其面向的读者却仅限于社会学家，而非人类学家。其中有几章是关于研究本身的，比如：“The Dynamics of Bureaucracy”、“Men Who Manage”以及“Union Democracy”——正在研究程序开发的心理学家们，应该会对这些章节感兴趣。

□ Webb, Eugine J., et al., *Unobtrusive Measures: Nonreactive Research in the Social Sciences*, Chicago, Rand McNally, 1966.

该书是一份关于心理学研究技术的特定子集的综述。

□ Sackman, Harold, *Man-Computer Problem Solving*, Princeton: Auerbach Publishers, 1970.

尽管其副标题为“Experimental Evaluation of Time Sharing and Batch Processing”（分时及批处理的实验分析），然而这却是运用心理学方法对程序开发过程进行研究的第一本

① Field Work，田野工作，也可称为田野调查。在社会学、人类学等领域中的一种研究方法，指的是深入社区（尤其是边远贫困地区），通过参与式观察与访谈等形式采集数据、了解情况，进而在理论上取得发展与创新。——译者注

书——就凭这一点，它也是每个认真的学生必读的一本书。

□ Hyman, H. H., *Survey Design and Analysis*, New York, Free Press, 1955.

就公众观念调查的理论及实践而言，这可能是最好和最全面的一本书。

□ Cronbach, L. J., *Essentials of Psychological Testing*, 3rd ed., New York, Harper and Row, 1970.

一本基础性的书籍，同时也是一本有用的导论，介绍了各类心理测试的构成方法及其应用。

□ Hammond, K. R., and J. E. Householder, *Introduction to the Statistical Method*, New York, Knopf, 1962.

在关于心理统计学的数百本著作之中，这是较好的一本，它相当于一本导论。数学基础好的程序员不一定要选择这本书，因为它面向的是那些没有太多数学基础的程序员——当然，其文笔非常优美。无论如何，了解心理学家研究定量数据的思路肯定是非常重要的。

□ Morgenstern, Oskar, *On the Accuracy of Economic Observations*, Princeton, Princeton University Press, 1963.

经济学也是一门行为科学，尽管它与程序开发心理学没有很多直接的关系，然而其在方法论方面的问题却可以使我们受益匪浅。Morgenstern 与 John von Neumann① 一道创立了博弈论，在这里，他敏锐地意识到并开始思考这样的问题：经济上的数字到底有多大用处？然而，该书前半部分的讨论，根本没有局限于经济学领域，因此对任何一位行为科学家而言，这部分内容都是必读的。

□ *Proceedings of the Nth Annual Computer Personnel Research Conference*, Association for Computing Machinery, New York.

随着 N（该会议之届次）的增长，该会议的规模以及涉及领域也在扩大。与此同时，该会议的论文集也更加容易获得——其中的内容是该领域内的每位研究人员都必须了解的。虽然其早期的会议论文集很难找到，但是其价值也许要超过后来的论文集。对于当时的那些远未解决的问题，我们现在已经老练多了，这些早期的论文集中记述了当时的人们为了把握这些问题而进行的努力。

□ Parsons, Henry M., The Scope of Human Factors in Computer-Based Data Processing Systems, *Human Factors*, 12, 2 (1970), pp. 165–175.

① 约翰·冯·诺尔曼（1903—1957）。匈牙利裔美国数学家，致力于研究过程论和控制论，存储程序计算机概念的提出者，被誉为“现代电子计算机之父”。——译者注

虽然本文中的大部分内容只与计算机非程序设计方面的问题相关，但是在《人类因素》（*Human Factors*）杂志上出现的这篇文章，或许标志着研究人类工程学的工程师们对程序设计产生兴趣的开端。非程序设计方面并非不重要——事实上远非如此——而仅仅是因为这些问题超出了本书的范围，这方面的综述恐怕需要 3～4 本书的篇幅。文中没有任何新的数据，但是给出了几篇关于已经完成的工作的参考文献。

第 3 章　如何研究程序设计

评　　注

“或许，程序开发的确是一项异常复杂的行为，以至于无从把握，所以在很大程度上，这个过程必然笼罩在神秘之中。”（p. 35）

四分之一个世纪的实践让我明白了，程序开发并不是一项很难研究的复杂行为；但如果你是在实验室中闭门造车，那么困难也许是不可避免的。关于如何研究程序开发的过程（或者更精确地讲，关于**我**如何研究程序开发的过程），我想我的观念已经有所变化。学术领域的进展非常重要，但是相对于我的思想更新速度而言，这些进展相对很慢。对我现在的客户们来说，学术方面的进展同样也难以跟得上客户的要求，因为这些客户所感兴趣的，是稳定而可行的办法，而未必是那些所谓的学术思想。

在这些年里我注意到，最优秀的程序员同时也是那些最善于自省的。如果他们发现做错了什么，他们会对导致这个结果的思维过程（或物理过程）进行检讨；然后，他们会采取一些相应的措施，对这个过程进行调整。这种被称为“根源分析”（Root-cause analysis）[①] 的方法，正是人们希望从 Weinberg 那里获得的，同时也是 Weinberg 希望让更多企业掌握的——然而不幸的是，这种方法还没有得到很多人的青睐。更多的人仍然喜欢使用“过失追究分析”的方法，这种方法的效果恰恰

① 一种以“主动智慧”和“自我诊断”为基础，力图在少量的人力介入情形下，修正系统问题的管理方法。——译者注

相反，它会诱导人们把引发问题的根源给隐藏起来。

很久以来，在各种专题研讨会或者论著之中，我的注意力主要集中在对行为的观察上。比如在我主办的问题求解领导才能研讨会（Problem-Solving Leadership Workshop）上，我就受到过指责——其原因是，我“只是”教会人们发现以前不曾发现的问题（或者说，通过不同的角度重新审视那些已经为人们所熟知的问题）。我意识到，一旦他们掌握了这种观察问题的新方法，聪明的人们都有能力根据其观察到的结果，修正发现的错误，同时继承正确的方法。

霍桑效应是科学观察的大敌，因为它会干扰观察者，使他们发挥出非正常的能力。但是多年以来，我已经学会利用霍桑效应，解决我的咨询业务中的实际问题。几年前曾经有过一个这方面的惊人例子：当时我正在旁听一个客户的技术评估会，以帮助其开发人员找出在自己的开发过程中出现问题的原因。这次评估会很有成效，但是参与者都承认，平常的情况并非如此。我问他们何以至此，他们的结论是：那是因为会议是在我的观察下进行的。由于我在场观察，所以一旦表现不佳，就会非常窘迫。因此，他们要去了我的照片，放大到真人尺寸，然后把我从上面剪下来，并且让这个“偶像”旁观此后的每次会议——这一招看起来似乎非常管用。

通过这次以及其他几次观察，我得出了管理心理学中的一条基本原则：关注手下工作的领导，将会取得更好的成绩。许多主管都希望他们手下的程序员向一个个代码模块一样地工作，那些曾经做过程序员的主管更是如此。这些主管认为，每个部下犹如一个小黑盒，只要输入要求，工作结果就可以源源不断地输出；根本无须任何观察，更不用说相互之间的交流沟通了[①]。很多负责软件开发的主管，就是不愿意与属下并肩工作——这里最主要的原因就是，他们从未接受过任何有关的培训，以至于他们根本不了解具体工作应该如何着手。

① 参见：Gerald M. Weinberg, “Overstructured Management of Software Engineering,” reprinted in *Software State-of-the-Art: Selected Papers*, eds. Tom DeMacro and Timothy Lister (New York: Dorset House Publishing, 1990), pp. 4—13.

同样地还是通过这种观察，我还发现了一些新的、内容丰富的信息来源——然而，这些信息渠道往往受人冷落。在对拨往几个软件公司的服务热线的电话记录进行分析之后，我发现了人类行为中一些奇怪的新模式。同时我也认识到，人们围绕计算机所发生行为的差异，要远比我在25年之前的估计更大。

但是就有关人类行为的实验的成本和难度而言，实际情况与我的预言是一致的。多数研究是在大学校园的环境中进行的；尽管这种方式令人陶醉，但是在实际的程序开发实践中，这种研究的成果所发挥的影响却很慢而且很小。这些工作可能非常重要，也可能的确应该得到承认。然而即使如此，这些研究人员还是应该通过这些工作获得哪怕是一丁半点有价值的信息。事实恰恰相反，各所大学传递给其学生们的信息，却越来越与这些大学自己的研究结果背道而驰。在参与实际项目时，这些受过此类大学式训练的开发人员，把自己的工作当成只需要一个学期就能完成的课程作业。他们一会儿思考，过一会儿又可能把项目忘得一干二净，然后突然在键盘上落指如飞——在这仓促间完成的急就篇，糊弄个及格还是绰绰有余的。在大学里完成的软件开发项目，通常都是不可维护、难以使用和难以测试的。开发专业软件的目的，不仅仅是用来显示其开发者对计算机科学的概念有多么深刻的领会。当然，也有些大学教授会对自己学生的工程训练提出更高要求——我对这些教授表示敬意。

和25年前的情况相同的是：来自不同个体的差异性，依然是破坏项目可预测性的致命因素。团队式程序开发方法具有自然的社会性，利用这种特性，我们可以对这种差异性进行中和调节——但是另一方面，这种调节也将使我们无法得到关于个体程序员的实验信息。在实际项目中由个体差异所造成的巨大代价，验证了我的有关预言：独立工作的个体既不是研究中的有效单元，也不是程序开发工作中的生产性成分。

在本章“心理学测量”一节，我曾引述了Maxwell的一句名言：“认知，就是测量”。然而，也许是为了给这句名言增加一点贵族的名望，

现在几乎所有的人都把这句话的某个版本归于开尔文男爵（Lord Kelvin）[①] 的名下。这种将 Lord Kelvin 与测量观念的重要性联系起来的企图，令我感到非常困惑——因为正是这位 Lord Kelvin，也曾经断言：飞机绝不可能飞上天。

人们对测量投入了很多的关注，但是却没有在理解程序设计发展的过程时进行足够的测量。相反地，人们为了不与之交流就能控制独立的程序员——也就是说，某些人根据自己的臆测设计出一些模式，然后强迫别人按照这种模式去工作——倒是设计了一些测量。然而这并不总是一个好主意。

也许，导致这种想法的原因就在于，有一种信念依然萦绕在我们的心头挥之不去，这种信念就是：对于任何一个给定的程序设计问题，存在而且只存在一个解——且只有教师才知道这个解是什么。虽然当前众多的培训，仍然是基于这样一个错误的信念进行的，但是我们正在逐渐改善。

> “只要有一点艺术性的豪迈和想像力的拓展，我们就可以认为计算机程序员们拥有自己的某种文化——一套他们共同拥有的信念和行为——而他们日常的举止，正是由这种文化决定的。”（p. 50）

我在 1971 年曾经这样写到。经过这么多年，无论是在事实上还是在人们头脑里的观念中，文化模式都的确已经出现了。在我的《质量软件管理》（*Quality Software Management*）系列丛书中，我已经确立起了软件文化的思想，因为我认为在今后的 25 年中，文化将是少数的几个重要思想之一。

① Lord Kelvin（1824—1907）。19 世纪英国卓越的物理学家。原名 William Thomson，由于铺设大西洋海底电缆有功，英国政府于 1866 年封他为爵士，后又于 1892 年封他为男爵，称为开尔文男爵，以后他就改名为开尔文。开尔文从事的科学活动涉及多个方面，其对物理学的主要贡献在电磁学和热力学方面。由他于 1848 年提出、于 1854 年修改的绝对热力学温标，在 100 年之后的 1954 年被确定为物理学的标准温标。他也是热力学第二定律的两个主要奠基人之一。——译者注

Chinker

第Ⅱ篇

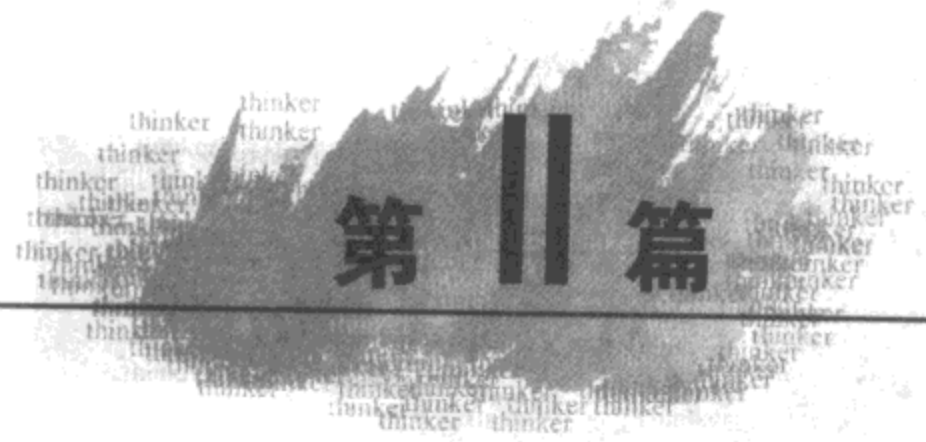
程序开发心理学

Computer Programming Psychology

第 4 章　程序开发组

第 5 章　程序开发团队

第 6 章　程序开发项目



作为社会行为的程序开发

出身于一个农奴主家庭的我，在初次踏上独立自主的人生之路时，与同时代的所有年轻人一样，对指挥、命令、斥责、惩罚等存在的必然性毫不怀疑。但是在刚开始的阶段，每当我需要管理事关重大的计划，每当我需要与各种人打交道，在任何的失误都会立即导致严重的后果时，我就开始意识到，建立在命令与纪律之基础上的行为，与建立在理解一致之基础上的行为是何等之不同。前者是一种酷似阅兵仪式的运行方式，但是对于真实的生命而言，这种方式毫无意义；只有通过后一种方式，将所有人的意愿都聚焦到一点之上，才会产生巨大的能量，并实现最终的目标。

——彼得·克鲁泡特金（Peter Kropotkin）*①

* Reprinted from *Memoirs of a Revolutionist*, by Peter Kropotkin. Copyright 1968. By permission of the Publisher, Horizon Press, New York.

① Peter Kropotkin (1842—1921)，俄国地理学家，政治哲学家，19 世纪末俄国著名的革命家。巴枯宁（Bakunin）于 1876 年去世之后，Kropotkin 便成为欧洲无政府主义的主要理论家和无政府主义者，对俄国和英国的无政府主义运动产生了很大的影响。他毫不妥协地反对目的可以将手段合理化的原则；他相信革命家个人的作为应该反应其人道理想，并毕其一生身体力行；他敏锐地注意到并且警告人们，以多数人之福祉为幌子但并非由多数人参与的革命，必然含有专制主义的危险；关于人类社会，他则试图建立一种条理一贯的“科学的”社会理论，认为改善人类现状的方法是合作而不是竞争——此处引述的这段话，正反映了他的这种理想。这段话出自 Kropotkin 的自传“革命家回忆录（Memoirs of a Revolutionist)”第三篇之第七节——经过在西伯利亚五年的实际体验，曾经一直对政府与纪律笃信不疑的 Kropotkin，开始重新审视自己关于“领导人与群众之间关系”的观念，由此“萌生出一种类似于托尔斯泰在其不朽之作《战争与和平》中所表达的思想”，并开始转变成为一名无政府主义者。——译者注

一般而言，程序员们并非孤军奋战。尽管每个程序员都可能会接到完成某个程序的任务，但是他总难免要向其他程序员寻求帮助；而其他程序员，也可能需要他的帮助。正如我们将要看到的，独自工作的程序员确实会困难重重。但是如果是与他人合作，他们之间会建立各种形式的关系。针对我们的研究目的，我们可以把程序员集体划分为三种类型——程序员组、团队以及工程项目。粗略地讲，这里讲的程序员组是若干名程序员的集合，他们在同一个地方工作，也许还共享同一台计算机或同一个操作系统；虽然有些程序之间会有某种联系，但是各位程序员却分工负责开发独立的程序。在各大学的计算中心、各工程公司或者任何一个提供开放式工作站的地方，都可能发现一个典型的程序开发组。

另一方面，虽然一个程序开发团队也是由若干名程序员构成，但是所有程序员需要互相协作，以共同完成同一项任务。一个团队的规模可能是2～12人；如果团队中的人数再多，就需要将它再细分为两个或多个团队。团队可能隶属于某个程序开发组，也可能由多个团队组成一个工程项目。而一个工程项目，除了包括若干名程序员外，还要加上一些支持机构，之所以要将所有这些组织到一起，可能是为了开发一个完整的系统，或者至少是关系密切的一组程序。它通常拥有自己的计算机，还有一些特殊的团队，这些团队负责提供系统工作、标准、文档以及其他服务。一个项目通常还有一位项目经理，以及一套日常管理的机构——就像奥地利军队[①]一样。

① 作者这里指的是古代的奥地利军队编制。直到19世纪，奥地利军队一直都沿用着中世纪创立的严格的封建军队建制。其实质性单位为团（Regiment），每个团分别由一位被称为Inhaber的贵族供养，这个贵族也就是该团的荣誉团长（Honorary Colonel）。通常还会有另一位贵族（被称为Second Inhaber），真正行使团长的职责。Inhaber对团内事务有绝对的领导权，包括对连长（Company Officer）的任免。在要进行大型战役的时候，由奥地利皇帝将不同的团临时组建成更大的军事建制，比如旅（Brigade）、军（Corps）。在计算机程序开发中，往往是按照项目（对应于战役）将实质性的开发团队（类似于团）组织起来，而且项目经理（对应于Inhaber）对各团队有绝对的领导权——这种方式与奥地利军队的建制非常相像。——译者注

第Ⅱ篇　作为社会行为的程序开发

评　　注

“我们可以把程序员集体划分为三种类型——程序员组、团队以及工程项目。粗略地讲，这里讲的程序员组是若干名程序员的集合，他们在同一个地方工作，也许还共享同一台计算机或同一个操作系统；虽然有些程序之间会有某种联系，但是各位程序员却分工负责开发独立的程序。……

另一方面，虽然一个程序开发团队也是由若干名程序员构成，但是所有程序员需要互相协作，以共同完成同一项任务。”（p. 63）

经过这 25 年，通过对很多程序开发组进行的观察，我对自己以前关于程序开发团队的定义做了调整。现在，我会说：“一支程序开发团队是若干程序员组成的集体，他们之所以以这种形式协同工作，是为了开发出更优秀的程序。”

换言之，要判断程序员的某个集体是否构成一支团队，要看其中的成员是以何种方式相互协作，以共同开发软件产品——这里所说的不是一般的软件产品，如果程序员是各自为战、缺乏协作，那么这类软件产品就不能被（或如此有效地）生产出来。不同团队所开发的产品可能是同一个；但是共同开发同一产品的多个开发组却不能称为团队。在有些团队中，每位成员所负责开发的产品各不相同（尽管这种情况很少见）；但是在许多开发组中，除了共享工作空间、受同一位经理主管外，各成员之间没有任何其他的共同之处。今天的硬件价格已经大大降低，与此

同时人力成本却不断攀升，因此开发组的成员已经不太可能像前些年那样，需要共享同一台计算机或系统了。

尽管研究团队与开发组之间的界限变得越来越模糊，但是归根到底，它们在某个方面还是有本质的区别——这就是它们进行学习的方式。25 年前的我对这一区别不太理解：既然也有一个共同目标，一个程序开发组内也应该会形成共同学习的风气呀？——从现在看来，很容易理解我当时的这一困惑。但是，在积累了更多实际经验之后的今天，我终于认识到：如果一个集体的共同目标仅限于产品的层次，那并不见得会促使其中的程序员互相学习。而反过来，团队内部的成员不仅目标一致，而且其目标与他们具体开发的产品毫无关系——正是在这种目标的引导下，一支团队的成员才会通过相互学习共同提高。

每年一度有我主办的软件工程管理组，就是根据这个目标组建的一些团队。Brian Pioreck 来自 1998 年的团队，他曾这样评价这个目标：

“有些企业所希望建立的是能够开发产品的团队，而不是能够开发产品的小组，我们的这一准则会对这些企业有什么帮助呢？他们将学会些什么呢？学会如何开发出更好的程序？学会如何更好地倾听意见？更好地交流？对自己感觉更好？我认为，所有这些问题的答案都是肯定的。

“如果我问客户：“在你们作为团队一员的经历中，那一段是最美好的回忆？”，他们最有可能的回答是：“那支团队简直就是一个家庭——其气氛就像是在感恩节前后一样。每个人都将自己创造出来的东西摆到桌子上，而所有人都可以来共享并为之庆祝。“我认为健康团队所具有的另一特点就是，它始终能够保持自我的生命活力。正是由于具有这种特点，即使有成员中途离队，也可以建立并维护一支具有共同价值观与习惯的新团队。”

第4章 程序开发组

对程序开发组的研究很重要，这可以帮助我们了解其他类型的程序开发组织形式。即使是将程序员组织成诸如团队或工程项目等正规形式，与所谓的“非结构化”集体一样，其中也会出现很多非正式的联系。确实，我们通过社会心理学需要首先了解的，就是正式与非正式集体之间的差别。

正式与非正式组织机构

主管们很喜欢玩组织结构图的游戏，但是如果将程序员之间的交流，严格限制为图中那狭窄而笔直的线条，那么程序开发将不会取得多少成果。也许是因为组织结构图与流程图非常酷似，所以经过层层选拔上来的开发主管们，似乎对这类图过分相信，而对不太正式的形式却不感兴趣——尽管在尚未出头的时候，他们曾经非常成功地运用过这些不太正式的机制。但是，人与人之间的交流沟通既不是那么狭窄，也不是那么直截了当，更不是用一张组织结构图就能表示出来的。而人们之所以会犯种种重大的错误，一个原因就在于他们总是认为：只有正式的结构，才是一家企业中惟一可能的结构。

当然，工程项目中的非正式结构在很大程度上要取决于其工作结构，因此或多或少都会与组织结构图相符；而具体符合到何种程度，则取决于该项目的组织化程度。然而如果是在计算中心，那么无论其正式结构如何，即使该中心只为程序员个人提供服务，也总会有一些相伴而生的非正

式结构来进行更正与补充。有时候，如果当权者非常明智，他们可能会把出自非正式结构的新方法以正式的形式固定下来——尽管不总是完全一样。

在一家咨询工程公司的计算中心里发生的一个故事，就是这样一个例子。那里的计算任务是以远程批处理方式进行的：在离开发组很近的地方有三座大楼，开发组内的大多数程序员就分散于其间。程序的周转时间很不规律，而且也没有什么可靠的方法能够在事先确定某个计算任务的完成时间。在该计算中心的输出服务窗口外设有一个公告牌，一旦某个程序运行结束，其对应的任务编号就会在上面公告出来。然而每当程序员前来察看公告时，却往往失望地发现自己的计算任务尚未完成；这样，如果其办公室距离计算中心很远，那么程序员每次动辄就得花上半个小时。

无巧不成书。有一位秘书的办公室位置绝佳，即使她坐在椅子上原处不动，也能够透过办公室的大门，正好看到公告牌。有一回，一位程序员一边等待程序运行结束，一边通过电话邀请这位年轻而极富魅力的秘书与他约会。一阵闲谈之后，秘书小姐提醒该程序员继续工作，他回答说自己正在等待一次计算的结果。“噢，”她调皮地说道，“你的任务编号是多少？我能从这儿看见公告栏。”这个消息不胫而走，没过多久，就连距离很远的那些程序员也知道了这项“服务业务”。实际上，这项服务简直太棒了；然而不久之后，秘书小姐就开始为完不成自己分内的工作而发愁了。这样，这种非正式的机制终于引起了管理层的注意。

这里的管理人员很明智，他们知道不能简单地废止这项服务，因此他们开通了一个查询电话：一旦拨通，程序员们就可以听到一个录音，向他们报告最新公布的任务编号。管理人员还采用了一个更聪明的做法，他们直接把查询电话号码设置为秘书小姐原来的分机号——尽管他们克服了来自电话公司的重重阻力，才将此付诸实施。就这样，一项不见得高效，但很有用的非正规服务被建立了起来，而且转换的过程非常顺利——这项措施的一个附带的好处就是，秘书小姐从此又可以有足够的时间来完成自己日常的工作了。

然而需要指出的是，新的正规系统并没有完全取代原来的非正规系统。因为每次录音需要经过 30 分钟左右，所以即使任务的编号已经在公

告牌上公布出来了，程序员也要经过一段（可能长达 30 分钟的）时间才能通过电话录音查询到。在通常情况下，这种效率已经足够高了。但是既然这是一个计算中心，那么总会有些人有特殊的需要。比如，有的程序员可能非常急于知道自己的计算任务的进展，在这种时候，他们就会转而求助于原来的系统——直接向秘书小姐询问。尽管如此，秘书小姐额外的工作负担已经大为减少，所以不至于对她的其他工作有什么影响，原来的系统也这样得以延续下来，而且非常可能会一直保留到现在。另外，有机会与这些年轻而聪明的程序员交谈，对她来说是一件**乐事**——当然我们也要承认，这些程序员也乐于与她交谈。

正规途径与非正规途径之间的此类矛盾，并非总能这样圆满地解决。在上述例子中，计算中心的管理人员能够充分认识到非正规渠道所发挥的作用；但是在其他方面他们却可能根本不会意识到这类渠道的作用。这方面的一个例子，来自于一所大学的大型计算中心。那里的大多数程序员为学生，他们没有自己的办公室，所以计算中心就在收发窗口旁，为他们提供了一间很大的公共休息室，以便学生们与其他程序员在那里研究各自的程序设计问题。在相邻的一间屋子，计算中心还同时设置了一个咨询服务站，雇佣了两名研究生助教，帮助学生们解答问题。

在公共休息室的一端，摆放了一排自动售货机——提供咖啡、可乐、糖果等等，应有尽有。这间屋子虽然面积很大，但是由于自动售货机旁经常聚集着一帮人边吃喝边玩闹，他们弄出来的声响很大，以至于一些程序员实在是忍无可忍。最后，有两名较计的学生自己成立了一个委员会，然后去找中心的主管反映意见。中心主管此前固然未曾大驾亲临过这间休息室，所以他决定亲自去考查一番。他一走进这间休息室，就立即被售货机那头的喧哗场面惊呆了。这次“考察”前后只花了不到 15 秒钟，主管就已经对此做出了“判断”。他旋即回到办公室，马上颁布了一道命令：把所有的自动售货机搬出休息室。

售货机被搬走才不到一个星期——当时强调环境肃静的告示已经是满天飞——就有另一批人来找中心主管。这批人的规模更大，组织得也更好。他们抱怨说，提供咨询的服务远远不够。当中心主管亲自前往实地察看时，他确实看到等待咨询的人排成了两条长龙，一直从咨询室伸展到公

共休息室。他问咨询员们，他们提供服务的速度为什么突然间会降下来了。咨询员则回答说，他们现在的工作虽然不见得更快，但是也决不会比以前任何时候更慢。他们还说，不知何故，前来寻求咨询服务的人比以往要多很多。

在接下来的两个星期里，该中心主管一直在试图找出咨询任务量陡增的原因。但是无论是课程还是用户，都与以前一样地正常运转。但是，咨询工作的负担依然很重；而学生们则开始抱怨说，他们可能会无法完成自己的作业。最后，中心主管派了一名主修社会学的研究生，深入到排队等待的学生之中进行调查。不久，该研究生就发现了问题的症结——那些自动售货机。

如果公共休息室中设有这些售货机，那么其周围总是会围着一大圈人——但是并不像中心主管在仓促之间所认定的，这些人在那里绝对不是游手好闲、无所事事。表面上看，他们确实是在一边喝咖啡一边高谈阔论；但是他们的谈话内容，却是有关正在编写的程序。实际上，学生们通常的行动规律是：来到中心后，首先到输出窗口取自己的打印结果，然后径直奔向公共休息室中的售货机。在那里，他们可以一边品着咖啡，一边初步浏览一下打印的程序；如果伙伴们也在身边，他就会给朋友们看看。由于学生们的问题一般都大同小异，每个学生都极有可能在售货机旁遇到另一位同学，可以帮助他指出错误所在。通过这种非正式的渠道，正式咨询机构的工作量就被分流了。就这样在无形中，实际的工作量大为减少，所以完全能够从容应对。

中心主管将自动售货机搬出之后，无意间破坏了这一非正规结构的中枢，于是原来被分流的那部分工作量又重新被导向咨询员。很可惜的是，顽固的中心主管怎么也不相信，这就是导致咨询工作量过多的原因。因此，他并没有命令重新安装售货机，而是一味地增派咨询员。但是这回又轮到那些咨询员开始抱怨了，因为他们需要解答的问题过于琐碎。最后，由于没有人愿意承担这项服务，所以该咨询站不得不关张大吉。我最后一次路过这个计算中心时，他们的咨询服务依然还是通过某些其他的非正规渠道进行的——学生们只有在各个办公室之间奔波几个来回，才有可能找到一位程序员能够回答他们那些琐碎的问题。最终的结果如

何，很难说得清楚；但是我们可以肯定的是，学生们的问题终归是可以找到答案的——不过，其代价必然不菲。

这则故事告诉我们：非正规的机制到处存在，而且如果你还没有真正搞清楚其规律，就企图改变什么，那将会十分危险——你可能会把有些操作系统搞得一团糟；同时，任何的替代方案都需要付出更高昂的代价。许多由此引发的系统紊乱，都源自于对环境的物理格局进行了贸然改动——在各个计算中心，这种改动非常普遍，因此我们值得进一步花费笔墨，讨论一下自然结构与社会结构之间的关系。

自然环境与社会结构

我们都知道，周围的自然环境，对我们工作成果的质量和数量都有影响。比如对于噪音、光线、温度等因素的影响，工业心理学中就有连篇累牍的研究成果，但是我们在这里的意思并不是指这些。在将程序员的工作场地整合到一处的过程中建立起了一些准则，毫无疑问，只要遵循这些准则，的确会带来巨大的好处（有些人所进行的深入研究显示，对工作场地做过于普通的划分，反而会对程序员的生产效率造成负面的影响。这些方法一方面会切断程序员之间有益的沟通渠道，而另一方面却为那些干扰性的声音、动作大开方便之门）。程序开发过程中的社会交流方式，会转而影响到正在进行中的开发工作。而我们所关心的是：工作场地的格局对这种社会交流的方式会有什么影响？

让我们来看一个简单的例子：将原来的老式电梯，更换为最新的自动电梯。对于程序员们来说，这是最不幸的一件事。实际上，旧式电梯的驾驶员还承担着一项非正式的任务——在进行程序开发的第八层与机房所在的地下室之间，负责接收和发送资料。当然，没有人能够让大家相信，仅仅是为了把资料从第八层送到地下室，就应该专门雇用一名信使。就这样，程序员们宝贵的许多开发时间白白浪费掉了。旧式电梯的另一个好处就是，驾驶员同时还能肩负起确定程序员在楼内位置的任务。如果机房设在某一层，穿孔机房设在另一层，而各程序员的办公室位于其他的某层，那么要是你希望只试一次就能找到自己需要寻找的人，其概率要低于

50%。但是如果有电梯驾驶员帮忙，你就可以立即知道你需要找的人现在第几层。还有其他服务的损失，比如重新送还误投的信件，以及重要信息的传递等等；虽然引入的新电梯本身速度会快些，但是即使只考虑前面的两个方面，也可以看得出这项措施是净亏损的。

即便说新旧电梯的更换可能很少见，但是在某些软件厂家，办公室的迁移却几乎是家常便饭的事。只要程序员们居于一处，就可以使他们意识到其他人的存在，这种效果很好。在以前，这一效果是由设在机房的休息室来实现的——因为所有人都要在那里排队，等候属于自己的短短 15 分钟的在线调试时段。我在 1956 年左右听到的一段对话，就反映了这种场合中最常见的情况：

“他在那里这么久，到底在干什么？我手头要完成的活可是至关重要的——现在就剩下最后一个错误要调试了。”

“他正在调试 FORTRAN。”

“FOR——什么？”

“FORTRAN，意思是指‘公式转换’。有人宣称，你可以像书写数学公式一样地编写程序，然后这个程序可以自动地替你把程序翻译成机器代码。”

“得啦，别胡扯啦。你不会是在打趣吧？”

“他们确实就是这样说的。依我看来，这应该可以办得到；但是其效率恐怕要比手工更低。所以，这玩意儿不会有人掏钱买的。”

“是的，这肯定会消耗大量的机器时间。看起来，他好像正在那儿弹风琴，上下翻飞地敲击着键盘。应该有更好的办法。”

“应该有。”

“哎呀，那是什么？”

“我现在正在使用这种更好的方法。我们正在西海岸进行这项工作，但是只有这里才有惟一的一台计算机，所以我们必须到这儿来进行调试。它的名字叫做监控系统。”

“监控系统？”

“是的，它有点像一位自动的操作员。可以让程序员从计算机旁脱出身来。比如，我可以把 30 项计算任务带到这里，并且能在短短的 15 分钟

内运行它们。”

“30项？你不是在拿我开心吧！”

“不是开玩笑，我说的是真的。不仅可以省掉操作员，我们还可以实现惊人的加速比——当然，设置与拆卸的时间也可以节省下来。”

“好吧，祝你好运。但是我还是无法想像，如果不在控制台面前，我如何才能对程序进行调试。像现在这样，我在这里等得都快要疯了——它名字是什么？——使用FORTRAN的伙计。再见，现在轮到我了。”

通过这种方式传递的有用信息，其数量是惊人的。正如自动电梯破坏了老式电梯对应的社会结构一样，操作系统也破坏了原有的社会结构。尽管如此，只要在领取计算机输出的地方，安排相邻的一间屋子作为公共休息室，那么人与人之间就会在信息交流的过程中，得到很多有益的东西。但是，个性化的邮件发送系统却会使程序员与这种交流隔离得更远；而通过终端系统实现的远程任务进入与退出，则会进一步地加剧这种隔离。从这方面来看，终端操作可能会成为一种灾难，而不是程序员的福音。

错误与惟我独尊

读到这里，很多程序员可能会感到困惑：为什么需要如此强调程序员之间的社会联系呢？程序开发是一项个人行为——也许与其他职业相比，程序开发的这一特点尤为明显。此外，这种个人的行为取决于程序员自身的、而不是其他人的个人能力。你每天遇到的其他程序员是多是少，对你的程序开发工作来讲会有何不同吗？如果被问及此类问题，多数的程序员会说，他们宁愿独自工作，而且其工作环境中最好不要有任何其他人的打扰。

上面一段中所罗列的一些观点，也许就是在改进程序开发方法的过程中，我们将遇到的最大的拦路虎。首先，如果这些的确就是人们对程序开发这一职业的普遍认识，那么人们对这一职业将会为之吸引、还是敬而远之，将只取决于他们是喜欢独立工作、还是喜欢与别人一道工作。社会心理学家告诉我们：不同人的性格特点也不尽相同——这一点似乎人人皆知，但是如果打上权威的标签，恐怕说出来的效果会更好。对于人们通常

的个性气质，有一种方法是通过三个方面来进行衡量——一个人或者是“服从的”，或者是“进取的”，或者是“超然的”。服从型的人之性格特点是，其态度倾向于“与他人合作，并且会对他人很有帮助”；进取型的人则总是希望“赢得金钱和声誉”；而超然型的人则倾向于“只有独自相处才能有创造力”。

实际上，每个人都是上述三种态度的混合体，只不过大多数人在某一种特点上的倾向性更重。毫无疑问的一点是，当今的程序员绝大多数都倾向于“超然的”类型；究其原因，一方面是他们个人的选择，同时也是因为现在雇用程序员的政策——这种政策常常会倾向于寻找这类人。而且在很大的程度上，这种选择还算不错——因为程序开发中的大部分工作，都属于“独立进行与创造性”的类型。

然而与其他的好事一样，程序员的“超然”经常被强调得过了头。尽管他们与其他人相互隔离，他们与程序的距离却更接近了。确实地，他们的程序常常就是他们自己的扩展——比如说在程序中附加个人姓名，这种令人生厌的风气，就是对这一点的很好证明——比如 Jules' Own Version of Algol[①]（其更为人熟知的一个名称是 JOVIAL）。其实，即使程序的作者没有被官方授予署名权，程序员也可以很清楚地知道，某段程序的作者是谁。

那么，保留对程序的“所有权”会有什么错呢？艺术家“拥有”他们的绘画作品；作家“拥有”他的著作；建筑师“拥有”他们设计的建筑。这些技艺精湛的劳动者，之所以可以陶醉于平庸之辈的赞赏和效仿之中，难道不正是这种特点的作用吗？在开始阅读之前，如果我们希望了解某本书的内容，封面上的作者署名难道不是非常有帮助的吗？进一步讲，这一点对于程序不也是同样适用吗？如果人们真的会阅读程序，那么也许是适用的，然而我们知道的是，很少有人这样做。因此，对于某个程序员的崇拜，不仅不会导致对他所编写的程序的效仿，相反却只会养成其怪僻的程序开发风格。正如我们在“艺术群落”中所看到的那样，每个人都知道如何打扮得像一位艺术家，但是很少有人能够像真正的艺术家那样动手画上

① Jules I. Schwartz 在 Algol 58 的基础上，于 1959～1960 年发明的一种程序设计语言，简称 JOVIAL。——译者注

一幅作品——即使有，恐怕也不会很多。

“基于特性的”程序开发方式真正面临的困难来自另一个方面。如果我们认为一个作品不入流，无论是绘画、小说还是建筑物，都无非是口味差异的问题。但是如果我们认定一个程序是不入流的——尽管我们已经知道，在判断“优秀的程序”这一问题后面隐藏着很多困难——这种结论至少是潜在地经过了客观的证明或证伪之后才得出的。最起码地，我们可以把程序交给计算机，然后检查一下其输出结果。面对关于其作品的批评意见，如果一位艺术家觉得不中听，他可以完全置若罔闻；但是，程序员又怎么能对来自计算机的评价置之不理呢？

表面上看，来自计算机的评价**似乎**是无可争辩的，而如果事情果真如此，程序员在自己的程序中署名的做法，必将严重地影响到他的自我形象。每当计算机查找出程序中的一个错误，这个程序员就肯定会这样来推理：

“这个程序有缺陷。由于这个程序是我的一部分，是我的扩展，其中甚至还有我的名字，所以**我**也有缺陷。”

这种自我鉴定令人感到刺耳，所以极少地被贯彻实行。针对被称为“认知失调”的心理现象，社会心理学家 Festinger 曾经进行过一些研究；在他之后，别人又进行了一些有趣的实验，目的在于揭示这一现象的本质。关于认知失调的一个经典的实验，是这样进行的：

> 被试们被分为两组，要求每人各写一篇短文，从正面来论述一下他们本来极不同意的某个观点。在第一组中，每写出一条理由，就可以获得 1 美元的奖励；而在另一组中，每条理由的奖励是 20 美元。在实验结束前，还要针对其观点，再次对每个人进行测试。我们也许会根据“常识”认定：相对于 1 美元的奖励，面对 20 美元的奖励人们更容易改变其观点。但是认知失调理论却预言：1 美元那个小组的成员才更容易改变其观点。这个预言，已经被数十次此类的实验所证实。

认知失调理论背后的道理非常简单。在上面简要介绍的那个实验中，两个小组的被试都必须完成相同的任务：写一篇驳斥自己所持观点的文章。在通常的情况下，他们都不会这样做——因为，试图去论证自己本不

相信的观点，这种行为无异于“伪善”或者“虚伪”，在我们生活的社会中，这两种行为都是受人鄙夷的。这样，就构成了一种失调的环境。一个人如果自认为是诚实的，那么此时此刻，其心中的自我形象就会与客观实际发生冲突——“我居然违心地写了这样一篇文章”。根据这个理论，失调对于人类而言，是一种不适的、不稳定的状态，因而人们必然会通过在两种态度之间进行抉择，以尽快脱离这种状态。为了脱离失调的状态，造成这种失调的某个因素必须让步——无论是这个还是那个，必须选择其一。到底哪个因素将会让步，这要取决于当时的条件。但是一般而言，不会是此人的自我形象——即使是最不可思议的论证，人的自我形象也能得以保留下来。

在刚才引述的实验中，可以获得 20 美元的被试，将很容易从其失调状态中解脱出来。“当然，”他们会这样对自己（或者任何可能提出质疑的人）解释说，“我并非真的相信那些观点。我之所以声称自己相信，完全是为了获得金钱。”为了金钱而去无端编造理由的行为，虽然不是一种令人起敬的美德，但是与质疑自己的信仰相比，却要好得多。但是让我们来看看，仅可以获得 1 美元的那个小组的成员们将会处于何等两难的境地。虽然参与心理学实验的被试几乎都是尚不富有的大学生，但是即使是对于他们而言，1 美元依然是微不足道的。于是，另一小组用来脱离失调状态的托词，对于他们就不再适用。因此为了从失调的处境中解脱出来，他们必须从其他方面寻找借口。至少对很多人来说，最容易的一个解决办法就是使自己承认：“我原先反对的这些观点，其中也确实有些道理”——只有这样自我解释，他们撰写这些短文的行为才不至于是虚伪的，而成为了一种目的在于培养公平和诚实意识的练习，而且这种意识可以使人们能够同时看到问题的两个方面。

在另一个应用中，认知失调理论可以准确地预言出，在做完一项大的交易（比如，购买了一辆轿车）之后，一个人的行为将会有何倾向。比如说有一个人购买了一辆福特轿车，如果你塞给他一大堆有关轿车的广告并要求他阅读，那么他就会把大部分时间用于阅读有关福特轿车的广告。如果他购买的是雪佛兰轿车，那么他就会对有关雪佛兰的那些广告更加关注。其实，人们一旦意识自己有可能进入认知失调的状态，就会有意识地

去回避那些可能导致认知失调的信息——刚才的实验就是这样的一个例子。因为如果购买的是一辆福特轿车，那么人们就绝对不愿意发现，事实上雪佛兰牌轿车更好；而避免发现这一情况的最好方法，就是不要看有关雪佛兰的广告。这样，无论看多少有关福特的广告，购买了福特轿车的人也不至于会发现原来自己并不是一个聪明的消费者。

现在，认知失调对我们的程序开发中的矛盾会产生什么影响，已经再明显不过了。如果一名程序员真的把自己编写的程序视为其自我的延伸，那么他就绝对不会希望在他的程序中发现任何错误。不仅如此，他还会尽力去证明，自己的程序都是正确的——甚至于，即使他发现了重大的错误，他也会睁一只眼闭一只眼，对错误视而不见。这种从认知失调中解脱出来的方法，实际上每个程序员对其现象都非常清楚——当然，仅仅是对人而不对己。当程序员带着他的输出打印清单穿过门厅时，这些文档很薄。如果无法隐藏程序运行过程中的错误，程序员就会像下面一样地加上一段注释：

“负责卡片穿孔的操作员又把事情搞砸了。”

或者，

“操作员把我卡片的次序弄错了。”

或者，

“什么时候我们的穿孔错误才能得到修正，并且可以正确地复制？”

这类抱怨有几千个花样，然而有一句话尽管非常简单，我们却从未听到过：

“我自己又弄砸了。”

当然，在错误很轻微、还不至于根本就没有输出的时候，程序员解决认知失调的方法甚至会更加简单——只要对其中的错误视而不见即可——其实，这些错误无论如何都是无法忽略的。假设其中没有任何错误：从某种意义上讲，人类的眼睛几乎是万能的——只要主人不希望看到某样东西，它们的确就可以视而不见、熟视无睹。那些精通于调试他人程序的程序员，可以逐一地举出成千上万的例子，来作为这个论断的佐证。如果注意力只是集中在他们自己的程序上，那么即使是输出中再明显不过的错误，也会被程序员忽略掉；而旁观者则可以一眼看出这些错误。所以，即使是

面对来自自然的、难以接受的反面证据，人们也往往会认定自己的程序完全正确。其实，这种倾向是再正常不过的了。如果我们想要解决阻碍程序开发质量提高的那些问题，如果我们想要从满足功能要求这一基本层次出发，那么我们将不得不对此有充分的认识。

无私式程序开发

针对上面提到的在程序开发中的惟我独尊问题，我们应该采取什么措施呢？管理学方面的资料通常都会建议，主管应该规劝手下的每一个程序员，要求他们加倍努力地查找自己程序中的错误。也许，主管每天都应该在程序员之间来回巡视，不时地要求某个程序员介绍一下他自己发现的程序错误。但是这种办法难以奏效，它带来的后果，与我们的心理学知识所讲的完全相反；因为，一般的人会把这种检查视为对个人的一种考察。此外，也不是每个程序员都有一位相应的经理——或者即使有主管，这位主管在看见某行程序被用红色突出显示后，至少应该能够明白是什么样的错误。

不，解决这个问题的方法，并非单刀直入式的着手处理——因为这样的进攻只会引起程序员的抵抗，而这种抵抗正是我们希望戒除的。确实，为了解决惟我独尊问题，必须重组社会环境；然后，通过这种方式，在这个环境中重新建立程序员们的价值体系。在讨论具体的方法之前，让我们先来看一些例子。这些例子将告诉我们，在采用这些方法之后，可能会发生什么情况——它们对程序员本人及其程序会有什么影响。

首先，请不要把这种重组看作是社会理论学家们象牙塔式的梦想。有一些程序开发组，确实已经克服了程序员的惟我独尊风气；实际上，远在计算刚刚成为可能之初，就已经有程序开发组能够解决这个问题。John von Neumann 很早就意识到，他在检查自己工作方面，确实能力不够，他也许是能够认识到这一点的第一位程序员。那些与他相识的人们回忆说，von Neumann 总是跟别人讲自己是多么蹩脚的程序员，并且不厌其烦地请别人阅读自己的程序，期望发现其中的错误与纰漏之处。然而在今天人们的心目中，von Neumann 一定是位无与伦比的计算天才——他浑身上下每

个毛孔都应该是完美无缺的。当然，von Neumann 的天才丝毫不容怀疑。然而他对自己作为一个人所具有的缺陷非常清楚，他的这种自知之明，也使他远远超过当今的一般程序员。

通过培训，凡夫俗子也能意识到并接受其自身人性的方面——亦即，他们无法与计算机相比的那些能力——并且对其做出正确的评价，从而在与同伴们共同工作过程中，能够尽量对此加以控制，以保证开发工作的圆满完成。Bill G. 是早期空间跟踪系统的一名成员。他负责的工作，是编写一个模拟器，对整个网络中的空间跟踪站以及实时的输入进行模拟。他的系统要求，即使没有连接到全球网，也能够实时地对系统中的其他部分进行检查。这个模拟器的核心，是一个非常微小而紧凑的循环；实际上，它只有 13 条机器指令。Bill 在这个循环上花费了很多时间后，自己终于觉得有些信心了，但他还是希望找到一些要求苛刻的人，对程序做吹毛求疵式的检查——在他所在的程序开发组，这是一项必须做的工作。

Bill 找到了 Marilyn B.，她愿意细读他的程序，他也细读她的程序，两人互相交换。在他们的程序开发组中，这是再平常不过的事；确实，没有人会在没有请旁人做这样仔细推敲的情况下，就急于到计算机上运行程序。只要有可能，就会以这种交换的方式进行互评，这样当事人就不会感觉到，自己正在受到别人的批评。但是对于 Bill 来说，由于他在这种方面已是训练有素，所以没有必要用交换的形式来保护自尊心。关于程序开发这一过程，他的价值观念认为，隐秘的、不与人分享的程序开发方式很不好，只有开放的、共享的程序开发方式才是好的。在他编写的（不是“他的”，这里的字典中没有这个词）程序中，可能发现的错误是人人可见的简单事实，暴露出这些错误，不过是为了将来更好的改进，所以并非对他个人的攻击。

在这个特定的例子中，Bill 经历了他的“程序开发生涯中糟糕的一天”。经过反复的检查，Marilyn 在他的代码中发现了很多错误；随着错误一个接一个地不断被发现，Bill 反而变得越来越开心。要是 Bill 受过的训练与我们大多数程序员一样，那他肯定早就与他们一样，开始忙着为自己开脱责任了。最后，在他们的一次学术会议上，他向全世界公布了一个令人瞠目的事实：在他短短的 13 行代码中，Marilyn 成功地找出了 17 个错

误。只要你愿意听，他就会不厌其烦地向你解释，这是怎样一回事。实际上，经过这次演习后，他只是认定那天并非他编写程序的“良辰吉日”。在那天剩下的时间里，他干脆把程序放到一边，然后一而再、再而三地到处向周围的人讲述这个小插曲中的每一个令人捧腹的细节。

而同时，Marilyn 并没有因为自己在这件事中起的作用，而感到暗自窃喜；她能够正确地对待这件事，她也能清醒地意识到：如果在程序中发现了 17 个错误，那么可能还有更多的错误。她特别清楚，虽然没有亲手起草这段代码，但是在对这段代码进行了相当长时间的修改之后，自己与 Bill 此前一样，已经把这段程序内在化了——也就是说，她自己的观念已经融入于其中了。因此，她同样需要找一位批评者。那天下班前，一边是 Bill 继续向同事们讲述这件快事，而在另一边，Marilyn 和其他人一起又发现了三个新的错误。

作为这个故事的尾声，我需要指出的是，当最后这段程序被加载到计算机上后，哪怕是经过任何“恶魔式的”测试，也没有再发现任何错误。实际上，这个模拟器已经被至少 10 个计算中心用于实时操作，并且在至少 9 年之内没有发现其他错误。对于每个发现的错误，Bill 并不是视之为对其自尊的伤害——事实上，这种自尊恰恰体现出一个人的愚蠢；否则，请设想一下，这个故事的结尾会是如何的不同。

这个故事并非只是一次孤立的事件，故事中的开发组也并非绝无仅有。然而，为什么这种程序开发组不是那么显山露水？为什么“无私式程序开发无私式程序开发”没有得以更加普遍地推广？人们似乎都认为这类开发组是凤毛麟角，大家之所以会有这种印象，与很多因素有关。首先，众多成功的软件公司，其实都建立在这种交流方式的基础之上，但是这些公司通常只是把这方面的知识看作是有价值的专有信息，除非你面对面地询问，他们才会承认。其次，采用这种方法的程序开发组都会极其自我满足和稳定，因此我们所见到的那些为工作而徘徊于各计算中心之间的程序员，不大可能来自于这类程序开发组。而且，为了保证工资能够持续地增加，这些“吉卜赛的”程序员必须制造这样的神话：只有天才的头脑，才能编写出最好的程序；除此之外，别无其他可能。

是的，没有任何人验证过，采用无私式程序开发方法与那些各自程序

员各自为战的方法相比，完成工作的质量的差别。这类方法之所以没有为更多人所知的另一个重要的原因。关于影响程序员生产效率的因素，虽然有很多实验做过研究，但是这些实验的设计大都存在缺陷——它们过于重视程序开发过程机械的一面，而忽略了其社会性的一面。比如，某项研究中可能会对分时系统与批处理系统进行比较，也可能对比一下 A 语言和 B 语言，其目的则很可能是，有人试图证明：他建立一个分时系统或者开发 B 语言的一个编译器的设想是合乎情理的。进行这些实验的人，似乎都想当然地认为：程序开发自然是以单独的个体为单位进行的。他们之所以这么认为，是因为这是他们通常最可能的工作方式。此外，对多个个体进行研究，问题会复杂得多。如果你在对 X 和 Y 两个系统进行比较分析之后，发现实验数据 90%的偏差都属于不同程序员个体之间的差异，那么有谁还会愿意徒增困难和成本，去研究程序开发组的表现呢？

如果程序员对他们各自需要实现的目标（高效率编写程序或者快速完成工作）理解不同，他们编写的程序也会很不一样。为了评估这种差异，我们进行过很多研究。在对方法进行讨论的一章中，我们曾经简要介绍过另一则有趣的轶事，这件事可以用来说明其中一个方面。与通常一样，我们雇用的被试都是相互独立的，实际上他们都是选了一门为期三个月的课程的学生。但是，其中有一名被试来自于一个采用无私式程序开发的开发组。有一次他前来找我，告诉我说，他的工作已经初具规模，现在是需要有人来对他的工作进行检查的时候了。鉴于这些实验的目的并非研究集体工作与个体工作之间的差异，我只好违背自己的信条，要求这位被试在没有外部协作的条件下继续工作——因为否则，就会引起实验数据中的偏差。

作为这个故事的一点间接说明，我们必须指出：虽然参加实验的程序员解决的是同一个问题，但是根据评估员的意见，这名程序员的工作，显然要比另外四个人的工作组织得更好，而且运行得也更好。当我们和他讨论这个问题时，他解释说，在实验期间他保持了他在自己原来的程序开发组里的一贯工作风格——始终注意保持程序的整洁和易于理解，以便将来终有一天别人来阅读这个程序。促使人们采用无私式程序开发方法的最初的也是最强烈的目的，仅仅是为了检查程序中的错误。然而透过这个例

子，我们可以领悟到，无私式程序开发方式的优点，还远远不止这些。实际上，参照好的程序开发方式所应具备的四个因素，依照该方法对这些因素的影响对我们自己做一检查，将是大有裨益的。

在满足功能定义方面，这种方法的价值显而易见。就开发进度而言，虽然其对程序完成平均时间的影响并非一目了然，但是从我们所举的调试模拟器的例子可以看出，其对程序完成时间偏差幅度的影响却是显然的。如果说在编写程序的过程中，程序员的运气是由日期决定的话（似乎很多资料都支持这一观点），那么在手背的那天编写出来的程序，就需要经过一个特别长的调试周期。就 Bill 的程序而言，可能需要好几个星期，才能将 20 个错误完全排除。而且，非常可能出现的情况是，即使在该程序已经与其他程序集成起来之后，还有至少一个错误会漏网并残留在系统内部——这将会造成系统其他部分开发的返工，或者至少其进度将难以预测。

不仅是调试时间的偏差幅度减少了，而且由于每段程序都有不止一个人熟悉，所以对实际的工作进展做出客观的估计，就是一件很容易的事情了。这样，就不必仅仅依赖于某个人的判断；否则，很难保证这一个人能够不带某种偏见。同时由于我们确定，至少有两个人能够理解该程序，所以程序的适应性也将得到提高。在某些特定的程序开发环境中，这意味着无限的提高。另外，在程序员中很可能发生的情况是，有关的某位程序员生病了、怀孕了或者找不到了。现在，即使出现这种情况，整个网络的运转也不容易受到影响。这不仅减少开发进度的偏差，而且在将来的某一天，一旦需要对程序进行修改，也会更容易找到一个了解它的人。

关于效率的问题，我们很难做出严格和迅速的说明。但是几乎可以肯定的是，通过这种方法所开发出来的程序，没有任何理由会比通过其他方式开发的程序效率要低。尽管程序整体的效率通常主要取决于最初确定的结构，但是通过请别人帮忙阅读程序，我们至少更有可能将最低效的方面消除掉。

这种方法的最后一个好处就是，它对阅读他人程序的人自己也有积极的影响——因为如果我们能够正确理解阅读程序的重要性，那么通过阅读他人程序这种实践，他必然会成为一位更出色的程序员。在有关程序员培

训的章节中，我们还将就此问题做进一步的探讨。即使缺乏专门的教育措施，我们仍确实可以看出，这些程序开发组的整体能力和水平会自行提高。

建立与维护程序开发的环境

如何营造如我们所描述的这样一个令人向往的环境呢？这个问题，与在该环境建立后，如何维护它的问题很不一样。相对而言，维护是一项容易得多的工作；而在建立这种环境的过程中，程序开发组现有的原则和观念需要转变，往往伴随着与其社会结构中诸如“锁定”或“凝固”等现象的斗争。某些条件所导致的环境，将反过来有利于该条件自身的保持，而一旦出现这种条件，凝固现象就会发生。比如一个调频广播的调谐器，可以将它设计成自动搜索到调谐旋钮范围内的最强信号处。一旦这个设备捕捉到了某个电台的频率，只有通过一个更强的变化才能解除它；否则，任何微小的变化都会由于调谐器的自动补偿而被抵消。这种锁定现象可能发生在任何系统中——生理系统、电子系统、生物系统，尤其是在我们正在研究的社会系统中。

社会系统中锁定现象的一个典型的例子，就是各个计算中心对某种语言的接受倾向。一旦某种语言已经被接受了，其他新的语言就很难再进入其中——因为只要多数人使用原来的语言，沿循大家常走的道路将会带来更多的好处：如果需要帮助，你可以更容易获得；如果需要子程序，你更容易找到现成的；使用大家常用的语言，计算机更易于制订运行的计划；由于提交的代码类似，打孔机出现的错误将更少；使用原有语言编写的程序，其开发过程更顺利，效果更好。

与在程序语言方面的凝固现象一样，在一个计算中心里同样会建立起一个整体的社会环境，对于无私式程序开发方式，这种环境要么积极提倡，要么一味抵制。当一位新的程序员加入到这个环境后，通过与其他“原住的”成员的交流，他对待问题的态度将会被渐渐地重新塑造。比如要是有一次他向别人请教，但是却因为其程序中的错误，被人耻笑为愚蠢，那么他恐怕就不会再尝试第二次了。反过来，如果有别人来找他，请

他帮助审阅自己的程序，那么这个人就会当着他的面，委婉而含蓄地对他的能力大加吹捧，于是在下次需要向别人请教的时候，曾经被奉承过的这个人心里就会有一种威胁感。在很大的程度上，我们的行为方式会在不知不觉中，趋同于我们周围人们的行为方式，因此，任何一个正在运转的程序开发组，都会倾向于将每个新加入的成员，纳入于该开发组的程序开发指导原则的轨道之上。

在有新成员加入的时候，程序开发组固有的指导原则自然会受到一些影响；但是有的时候，他们还需要面对来自其他方面的威胁，这类威胁要远比由一两个新成员构成的威胁更大。来自组织内部领导高层的干涉，总是不断地威胁到无私式程序开发的原则。主管们倾向通过“积极进取的”方式脱颖而出，同时他们也不会承认这样一个事实：在对金钱和名誉的追求方面，其他人的观念与他们并不完全一致。本来，只要在对个人才能的互相尊重以及为了共同的事业而进行合作的基础上，程序开发组就可以顺利地开展工作。然而，这些却尤其令主管们不知所措，因为他们不能理解其原因何在。恰恰相反，他们总是认为别人都和他们一样——之所以要工作，仅仅是为了金钱，或者是由于感到威胁。

在程序开发组中，积极进取的主管与温顺服从的或超然的程序员之间，常常会发生一些冲突。在一家计算机制造公司的软件部门发生的一件事，就是一个特别恰当的例子。这个部门的一个程序开发组一直担当着团队的任务，在开发某个全新的、具有很大市场销售潜力的系统时，该开发组取得了非凡的成功。鉴于该开发组的成绩是如此显著，公司的领导层决定为该开发组颁发奖金。根据通常的管理模式，他们要把这笔奖金颁发给领导该开发组的主管。但是这名主管却告诉上级领导，除非这笔奖金是颁发给全体成员，否则他不会接受这笔奖金。你可能设想得出来，当时上级领导们会对他的这一做法如何不解。

其实，只要他们能认识到，这项工作的成绩是属于整个开发组这一集体的，就会认为该主管的反应非常正确，而不是无法理解的。然而有的领导居然认定，这只不过是他为谋求更大数额奖金而使的一个花招；而有的领导则猜测，他领导的这个程序开发组正在变成一个“狂妄自大的”集体。无论如何，他们还是勒令这名主管接受这笔奖金，同时也促使该开发

组解体——这种想法实在有些龌龊。主管领到奖金后，立即平分给开发组中的每个成员；然后，该开发组全体人员一同离开了该公司，转而投向另一家独立的软件公司。

在这次事件中，该开发组通过在接受奖金后辞职来保护自身，以避免遭受来自外界的威胁。要是公司的高层领导们能够懂得应该给该开发组什么样的奖励，或者要是他们在处理问题时更加灵活，完全应该可以找出一个可行的解决方案，并且使该开发组能够对其他开发组的工作产生有益的积极影响。但是高高在上的领导们总是认为：任何工作之所以能够完成，完全是由于某位领导出众能力的直接结果，因此，这些领导很难找到解决此类冲突的办法。即使有些领导能够承认工作是由整个开发组完成的，但是在其他方面却又不一致——比如，他们不会认为开发组的产品是整个开发组共同的财产，更不会认为这个产品中凝聚了每个成员的汗水和心血。

在另一个有趣的例子中，有一个由10名程序员组成的开发组，集体加入到一家制造飞机机身的大型公司，隶属于其中的一个程序开发部门。在此之前，他们已经共同为另一家公司工作了2年，但是那家公司却准备将其数据处理部门，搬迁集中到国内很远的另一头。这些程序员并不愿意搬迁。他们每人都得到了好几家公司提供的邀请，而且待遇丰厚，他们最后之所以决定前往这家公司，是因为该公司愿意雇用他们中的所有人。程序开发组的这种大规模集体搬迁，并不是很少见的事情。尽管高层领导们往往认定这是他们集体玩弄的阴谋诡计，但是他们的目的并不是希望借此获得巨大的物质利益。实际上，他们只不过是为了一个愿望能够得到满足：大家可以继续共同工作——有些工作环境之所以极具吸引力，其强大的魅力正在于此。

就在这个开发组投向这家飞机机身制造公司的几个月后，在一次计算机会议上，该开发组的新任主管遇到了它的前任主管，并向这位前任主管询问，他手下现在是否还有更多这类优秀的程序员。在得知整个开发组都已经加入了自己的公司之后，新任主管又接着询问，能招募到这些程序员的诀窍是什么。前任主管怎么也想不出当时有什么特别的之处——其中的一位女士是主修意大利语的应届毕业生，一位先生在一所高中教了7年的数学，另一位先生是专业的工程师，还有一位女士毕业于一所商学院，并

从事执行秘书和会计工作有好些年了。前任主管感到不解：你为什么会问这些问题呢？

“我只是不清楚这些问题，”的确非常困惑的新主管回答道：“但是自从他们加入我们的部门，我发现只要有任务需要保质保量、按时完成，我就会交给他们中任何一个人。我的手下另外还有300名程序员，但是如果我希望事情能办好，我却只能委托给这个开发组。这些人应该都是些天才。”很显然，这位新任主管的看法带有其浓重的固有观念的色彩，正因为如此，他无法参透一个简单的道理：每次当他给一个成员分派了任务后，他们都会按照自己通常的方式，由整个开发组共同来完成。但是当这个开发组向主管解释这个道理时，他就是不相信——他总是错误地认为：工作肯定是由其中的个别人完成的，但是开发组成员企图掩盖这个事实，因此在汇报的时候把其他人也捎上了。所幸的是，该主管没有因为搞不清楚该开发组工作如此令人满意的原因，就强行拆散他们。但不幸的是，由于他无法领悟出该开发组的成功之道，所以错过了将这种方法推广到其他开发组的机会。也因为如此，该开发组得以保留下来，虽然对外界仍然犹如一个封闭的口袋。最后，该开发组再次全体跳槽——当然，这次是一个更能理解他们的新环境。

当然，那些在程序开发过程中秉承个人主义原则的开发组，同样有其保持自身风格的办法——正如那个飞机机身制造公司的程序开发部门中的其他开发组一样。实际上，在一个如此大型的程序开发组中，任何一个新成员，甚至一个新的开发组，无论对自己的做事方法如何自信，都不可能改变整个开发组原有的社会系统；即使是其中的一个新的子开发组，也没有这种机会。如果他所加入的开发组成立已有时日，那么很可能他最终会放弃自己的方式，渐渐转向大多数人的方式——尽管他需要为此承受来自心理方面的痛苦，这种痛苦之严重，以至于没有人愿意承受。但是在更多时候，程序开发组可能新近才成立，或者是临时由多个独立的部分拼凑起来的。这种时候，其中的成员可能还会努力争取，企图根据自己的观念来塑造整个开发组。当然，其结果往往是他不得不悻悻地卷起铺盖走人。

还有一个与此对应的例子。其中的主人公名叫Jim A.，他在纽约的一个计算中心工作。有一次，他被调往一个在芝加哥刚成立的项目。他被分

配到的那个开发组有两名主管，他们都是成长于有良好的无私式程序开发传统的环境，而且他们决定在这个项目中推广这种方法。这个开发组中除了这两名主管和 Jim 外，还有四名实习生。在开发组成立的当天，两名主管就开始向其他成员鼓吹他们的工作方法。他们提出，从第一天起，无论是哪一位成员完成的哪一项任务，在上机运行程序之前都必须至少经过另一名同事的签字。他们希望通过这种稍稍正式一点的办法，来保证开发组成员能够养成一些好习惯，比如在工作中做到未雨绸缪。

在第一天的会议上，Jim 一言未发。但是等到实习生们离开后，他对开发组主管说："你们的想法很有意思，这可以帮助那些实习生掌握窍门。"

"是的，"主管们很有耐心地解释道，"但这些规定并非只限于实习生，而是适用于我们每个人，这样我们才不至于一开始就染上坏习惯。"

"你们还是不要那么认真吧，"Jim 笑道，"我的经历比你们要多两年。所以，根本不需要任何人来审阅我的工作。难道，我还需要这些实习生来教导？"

和众多的预言一样，这句话最后居然也兑现了——Jim 竭尽全力地不让任何人通过任何方式看到他那"神圣的"程序。没过多久，他在开发组中明显地就只有帮倒忙的份了。看到实习生们不断地向难度更大、更有挑战性的任务进发，而自己只能在角落里孤军奋战，他只能一味地讥笑他们缺乏独立工作的精神与独立思考的能力。他编写的程序，在质量上与组内其他人的工作距离越来越远。最后有一次，主管终于忍无可忍，把他的一个程序转给一名实习生进行整理；但是，Jim 宣称该程序已经调试完毕（或者"基本上"调试完毕），就是硬撑着不认输。

在这个例子中，这个开发组的社会环境非常强大，足以改变实习生们的工作方式；但是，它还没有强大到足以抵消 Jim 的"两年经验"。另一方面，Jim 的个性也不是很强，还不足以根据他自己的观念，将整个开发组引向他自己心中的方向。因此，最终的结局只能是令人难以忍受。也许，如果该开发组的主管更聪明或者更有经验一些，他们就会在一开始就把 Jim 清除出去，但为了这"两年经验"，他们不得不咽下失败的苦果——在其他领域，非常缺乏经验的人也会由于对经验的渴望而导致这类问题。

小结

程序员的工作环境多样而复杂，其间人与人的关系错综复杂，充满变数，稍不留神就会引起误解。如果要了解这种环境，你必须首先对正式与非正式结构加以区别，同时要找出决定和造成这种环境的众多因素：从周边的自然环境，到个人的自私性。任何一个正在运转中的程序开发环境中都包含了丰富的内容，这种丰富性使得它具有一种自我保持的特性，这使得该环境可以抵挡来自外部的强制影响——尤其是那些不顾正式与非正式之间的差别，就草率做出的强制性命令。在任何社会层面上，都能看到这种自我保持的现象；从本质上讲，这无所谓好，也无所谓坏。它仅仅是程序开发过程中的一种客观事实。

思考题

请主管回答：

1. 你是否有一张组织结构图，用来显示你的属下或与你平级的机构设置？找到一份这样的图表，然后用波浪线标出在你所在的企业中存在的人际交互关系。这些波浪线与原来的直线条吻合吗？如果是，请离开你的座位，到实际中去查看一下，那里的实际情况到底如何。

2. 你上一次要求手下更换办公地点是什么时候？除了你对搬迁所预计的后果之外，你还能想起来属下有什么行为上的变化吗？如果今后你再准备搬迁，你的做法会有什么不同？

3. 回顾一下你与程序员们的交流过程，你能想起曾经说过哪些话，有可能使得他们进入认知失调的境地（这时他们不得不采取措施保护其自我意识）吗？在这种情况下，导致失调的决定是否总是与你的意愿一

致？或者说，你是否有这样的体验：他们不是在改正程序中的错误或进度的延迟，而是在尽力掩盖这些问题？你应该如何深入研究这些情况，以减少失调的影响，或者把导致失调的决定，导向更加有利于你所领导的组织的总体目标上来。

4. 为了将无私式程序开发方法引入到你所领导的开发组，你会怎样做？你预计将遇到哪些阻力？你会如何克服这些阻力？你预计这需要持续多久？成功的可能性有多大？

5. 对于你所领导的单位中那些不思“进取”的人，老实地讲，你会如何看待他们？对于自己的工作很满意，对自己的收入也很满意的，都是哪些人？你对这些人的看法，在多大程度上受到你自己情感的影响？

请程序员回答：

1. 如果在一个计算中心，其解题的周期是稳定一致的，那么就不需要有非正式的形式来提供关于何时计算任务完成的信息。由于程序开发的复杂度所导致的偏差，将在多大的程度上影响非正式社会结构的发展？请根据你自身的经验，给出一些实例。

2. 如果你有规律地使用一个终端系统，你如何与该系统中的其他用户交换信息？你所使用的终端系统中，是否提供了一种操作，使得你可以与其他的终端用户交换消息？如果有，那么与你所使用的其他方法相反，这种机制对实时通讯有多大的价值？

3. 在谈及自己编写的程序时，你是否会称之为“我的”程序？请尝试在为期一周的时间内，在引用程序时不再使用人称所有格；然后，将你观察到的效果记录下来。

4. 你是否会将在“你的”程序中所出现的错误归咎于其他的人？你是否还曾指责过某些无生命的对象，比如卡片穿孔或者磁带？你的此类怨天尤人，有多少次是正确的？

5. 在发现“你的”程序中的错误时，你是否诅咒过“手气太坏”？

这种情况有多频繁？其他的程序员也和你一样的倒霉嘛？如果不是，你觉得为什么命运偏偏要专门捉弄你？你需要如何祭拜它们，才能平息这些神灵对你的怨怒？

参考资料

- Lynch, Kevin, *The Image of the City*, Cambridge, M. I. T. Press, 1960.

 Lynch 在这本以小见大的书中所研究的，是我们对周围自然环境的印象将如何对我们的生活产生影响。虽然这本书中的讨论只限于城市，但是无论你是需要改变或是要保留我们工作所处的自然环境，这本书都会为你提供丰富的信息资源。

- Wright, Frank Lloyd, *An Organic Architecture: The Architecture of Democracy*, London, Percy Lund, Humphries & Co., Ltd., 1939.

 美国最杰出的建筑师 Wright，在这本书中详细地介绍了他的“形式服务于功能”的理论：应该根据其需要完成的任务，来设计自然环境——这就好比是一套衣服，必须根据穿衣人来量体裁衣。虽然在今天看来 Wright 的思想已经显得过于僵化和刻板，但是在衣服和建筑都是现成的时代，感觉敏锐的主管经理在稍微读读 Wright 的书后，也许对自然环境的看法会有所改变。

- Goffman, Erving, *The Presentation of Self in Everyday Life*, Garden City, N. Y. Doubleday, 1959.

 工人们的行为方式是由其自然环境所塑造的，同样地，也是由社会环境的景象塑造的。然而如果是一个工作组，它同时也会对其社会景象有所影响，在认识其中的真实结构的过程中，主管经理们之所以要经历许多痛苦，这就是其中的原因之一。为了向外界以及新加入者展现其社会形象，单独的个体以及群体需要对其社会形象重新塑造，Goffman 对这种塑造所进行的方式（主管经理们与程序员们对此应该有所了解）做了研究和探索。

- Hall, Edward T., *The Silent Language*, Garden City, N. Y., Doubleday, 1959.

 通过一种与众不同的视角，Hall 对社会机构的形象重新进行了审视，并进而对于我们认识我们所在的工作组中的正式与非正式结构的方法进行了描述。程序员们的社会化方式，以及我们认识特定程序开发实践的方式，都无法从教材与书籍中学到。关于这些问题，Hall 的方法却可以使我们得到很多领悟。

- Festinger, L. A., *A Theory of Cognitive Dissonance*, Evanston, Ⅲ., Row, Peterson, 1957.

一个工作组预言了一个世界末日，但是等到那天来临时，却发现这个世界依然如故——在这种情况下，他们会如何处之呢？Festinger关于认知失调的研究，正是发端于针对这类（预言失败）问题的早期研究。在某些环境中，每天都有预言（尤其是那些诸如“这回程序肯定能够运行了”、“现在最后一个错误已经解决了”之类的承诺）失败事件发生；很显然地，在这种环境之中工作的人们，可以从认知失调理论理论中学到很多东西。

□ Haire, Mason, *Psychology in Management*, 2nd ed., New York, McGraw-Hill, 1964.

许多主管经理伤透了脑筋，也弄不明白应该如何去调动主管以外的那些人的积极性；对于这些主管经理们来说，Haire写的这本小巧而精致的书，是一本很好的启蒙教材。

□ Fano, R. M., and Corbato, F. J., Time-Sharing on Computers, *Scientific American*, 215 (1966), pp. 128-140.

在这篇非常流行的文章中，作者使用的方法与众不同。对于那些成长于特定的计算机环境中的用户，作者并没有急于给出任何有关这些用户所构成的社会分析的线索。也许恰恰相反，他们认为对于专业的读者来说，这并不是一个适宜的主题，所以他们根本没有沿袭这种方法。

第4章　程序开发组

评　　注

今天，批处理式计算中心大多已经成为历史博物馆，尽管如此，自然式公共中心的社会功能始终存在。但是时至今日，电子邮件已经无所不在，而在25年前，这根本没有（当然，我也没有能够预言到它的出现）。但是，电子邮件无法替代人与人之间所有的交流功能。电子邮件这种交流方式的一个好处，就是其相对的不可见性——这样，那些疑神疑鬼的主管们如果无法理解某种通讯渠道，他们就不再能够轻易地破坏它了。但是，偏执的心理仍然大行其道，仍然有些主管会通过精心设计的系统，对电子邮件进行监听。或者也许，这种现象不能简单地归为偏执，而不过是窥阴癖。无论如何，我对变态心理学一直都不在行。

工作空间被不断划分成小隔间的情况仍在继续，如果说有什么可以证明在位的主管们没有把心理学的研究放在心上的话，这就是一个有力的例证。我原来以为，DeMacro 和 Lister 在他们的《人件》(*Peopleware*)[①] 中向人们阐述过这个问题之后，它们就应该不会再出现了。也就是说，要是我不曾了解到管理心理学将社会地位凌驾于生产效率之上的做法，我确实会这样认为。Brain Pioreck 对这种现象是这样评价的：

> 我确信，这个问题较之社会地位更为深刻。我搞不懂为什么《人件》会成为一本畅销书……在我所遇到的主管当中，很少有人

① Tom DeMacro and Timothy R. Lister, *Peopleware: Productive Projects and Teams* (New York: Dorset House Publishing, 1987). 本书第2版的中文版已由清华大学出版社出版。

通过阅读而对其产业、管理理论或者心理学有所了解，根本没有。在过去曾经相信，关于自己的专业工作，他们所掌握的信息已经足够多了；但是坦诚地讲，就其工作的要求而言，主管们仍然缺乏培训，或者是自我培训。就社会地位而言，IT行业的主管们本身就是开发人员，他们在所就职的组织中已经被剥夺了任何地位，所以与来自其他领域的同级人员相比，这对于他们将更为重要。我弄不明白，为什么有些信息主管总是热衷于在办公室的角落里为自己建造一个个人王国。

与许多国王一样，主管们也会通过分而治之的策略来管理手下的臣民。但是，每个程序员都离不开与其他程序员之间的沟通。我在1971年曾经预言：由于远程系统的改进和推广，程序员之间缺乏沟通的问题将越发严重。当时，我只说对了一半。在一些管理糟糕的组织中，程序员日益感到孤独的情况确实会出现，但是我却过于低估了人类对与同类沟通的渴望。程序员们也许不会愿意在一家他们本不该呆的公司里虚度光阴，但是他们总是能够找到充分的理由面对面地聚到一起。

时至今日，我仍然同意我曾在《程序开发心理学》中就错误及惟我独尊的问题进行的论述；但是关于对人性的描述，我不再使用那些稍有审判意味的术语：顺从的—进取的—超然的。我现在更倾向于使用的，是更具描述能力的类型指数（Myers-Briggs Type Indicator，MBTI）；这是对人的性格的一种编目系统，在25年前，它尚不为公众所知——当然，这也包括我在内。关于程序员与主管经理互动的行为趋向，我在《质量软件管理》系列丛书中有部分论述；我发现，在理解这些行为趋向时，MBTI系统很有帮助。

在第1版中介绍的所有概念中，在本章中首次被描述的无私式程序开发概念，可能是被引用得最多的，同时也是最经常被误解的，也因此招致了最多的非议。我曾经常考虑，自己是否应该把这段写得更具说服力。如果我把这个术语替换为“更不自私的程序开发”，也许就不会引起那么多的争论了。或许我应该引用更多、更好的实例。或许，我应该

通过实验得到更多证据。

> “是的，没有任何人验证过，采用无私式程序开发方法，与那些各自程序员各自为战的方法相比，完成工作的质量的差别。”（p. 80）

在论述无私式程序开发时，我是这样写的。经过25年的岁月，这个方面的情况已经有所改变，这方面的证据现在比比皆是，比如：通过技术复查（审查或者其他形式），可以开发出更加可靠的代码，而且其成本会更低，一致性更好[①]。另外，的确，作为开发过程的一个标准的组成部分，现在有更多的软件开发组织有规律地应用各种形式的技术复查。但是并非全部——无论从何种意义上讲。

所以，我已经学会一些东西：如果你不希望自己的程序接受别人的检查，而又说不出什么合乎逻辑的理由，那么就无法通过逻辑论证的办法来改变你的工作方式。另外，如果你在自己的专业工作中根本不使用逻辑，那么你的专业水平必然会受到极大的限制。

不幸的是，关于建立与维护程序开发环境的论述材料虽然已有几十年的高龄，但是在今天看来仍没有丝毫的过时。有些主管仍然会与高效的程序开发团队发生冲突，并进而又以各种毫不相干的借口，将这些团队解散。而有些主管虽然还是搞不清楚这些团队的运行方式，但是他们却至少懂得应该容忍这些团队的存在。另外，关于团队集体在软件开发过程中的价值，还有个别人仍然没有认识到。

或者，也许他们确实认识到了，但是受制于其对个人职位的患得患失，他们的做法总是南辕北辙。现在对软件开发主管们的考评，很大程度上仍然是根据其已有的成果多少，而不是看其是否有能力建立能够创造出更多成果的团队，或是看他们的成果质量。在这样的压力下，主管们就会极尽其能事，编造种种相互矛盾的甜言蜜语去哄骗其属下，以期得到更多成果——当然，这些成果大多是短期的。他们的一些下属也可

① 有关内容请参阅：Daniel P. Freedman and Gerald M. Weinberg, *Handbook of Walkthroughs, Inspections, and Technical Reviews*, 3rd ed. (New York: Dorset House Publishing, 1990)。

能领悟出这个游戏的原理，于是也会效仿起这种伎俩，去欺骗他们的下属……；这样一直下去，直到他们也成为这样的主管。这种主管根本不屑于去建立团队，也开发不出什么高质量的成果，而只会再带出更多像他一样的下属——有朝一日这些人又将成为下一代被这样误导的主管。

第5章 程序开发团队

随着待开发系统的规模日益增大，在程序开发方面经验匮乏的现象越来越明显。虽然我们已经认识到，现实中的程序员与理想中的状态不同，他们并非是在孤军奋战；即使是很小规模的问题，在小程序和大程序之间也存在社会性的差异。如果一个人就足以应对整个开发组的工作——无论是工作目标、总体结构，还是最终代码的细节——那么程序开发的协调工作就没有必要。有时，两个程序员进行审查的程序，由他们独自就可以完成；而有时，由于整个系统过于庞大，任何个人都无法单独完成，于是他们只能各自完成整个系统的一部分。在前一种情况下程序员之间的交互，与后一种情况下的交互是截然不同的。这种差异，将体现在不同需求之间发生矛盾时问题的得以解决的方式上。在前一种情况下，冲突的解决是通过一个人的思考过程实现的——虽然可能会得到其他人的帮助，但是这个人始终拥有主宰权。在后一种情况下，必须首先透过技术上互相矛盾的需求，发现其背后的人际冲突，才能进而通过一种社会化的机制，来加以化解。

团队的组建

当然，一支程序开发团队之所以成立，是为了承担并完成某项由任何个人都无法独自完成的任务。这种要求，不仅仅与待完成工作的技术要求

相关，同时也与可以找到来参加该项工作的人员的能力以及允许完成任务的时间长短有关。为了保证工作的可行性，这两方面的因素——无论是团队成员的个人能力，还是允许时间的长短——都必须做到最小化。比如，如果团队的成员都是实习生，那么无论其规模有多大，有些程序开发工作必然是无法完成的。在商业上，这类人常常被称为“滥竽充数者”；而无论他们的数量如何加倍，对于工作的完成而言总是无济于事的。任务进展的日程也有类似的限制——为了理解这一点，你只要假想式地做这样一次试验：如果在某位妇女的基础上再增加 9 名妇女，让她们组成一支团队，那么是不是只需要一个月的时间，就能够完成生孩子的任务呢？

从概念上讲，任何系统的完成都会在经验值和时间方面有一个起码的最低要求。由于无法用数字精确地对其进行定义，同时也是因为对程序进行估计的不确定性，主管们在建立一支团队后经常会发现，根据某些判断可以显示出，所指定的任务根本无法在指定的时间内完成。于是不可避免地，在预定的期限到了后，我们不得不面对现实，给该团队再追加一定的时间。如果能够在一开始就学会面对现实，那么我们就会做更长远的规划，对工作的组织以及推行的方式就可能很不一样，并且到头来反而完成得更快。

在程序开发中司空见惯的这种情况，说明了在个人能力与日程进度二者之间存在着互补的对应关系。只有把工作交给最出色的团队，才有可能使开发的日程尽可能最短；另一方面，只有在愿意把开发日程拖得很长时，才可能使得团队的规模达到最小。换而言之，只要我们允许开发日程拖得更长，即使团队的编程能力弱一些，也能完成几乎任何程序——当然，编程能力不能降到基本限度以下。

这种情况也显示出，在开发日程与工作结构之间同样存在着一种重要的关系。我们把等到日程超出后再临时匆匆延长日程的做法也称为是一种帕金森病（Parkinsonianism）。虽然我们不得不始终对此保持警觉，但是过于紧凑的日程进度最终必然会诱使人们去寻找捷径。虽然采用这些取巧的办法或许可以保证目前这一任务的按时完成，但是其前提是所有的子任务都按计划完成，然而这种情况非常罕见。因此，许多程序开发任务之所以没有在计划的日程内按时完成，其原因都可以追溯到最初的日程规划以

及开发路线上，最初的这些计划总是一厢情愿地把开发过程中的所有条件都设想为最好——忽略了人生病、计算机故障、编译器错误以及所谓“绝对不可能发生的”错误等等可能。我们确实可能曾经有过这样幸运的经历，比如：在整整半年之内，可能团队中所有人的身体都极其健康；也可能计算机一直运行正常；也可能编译器没有出现任何错误；也可能的确没有出现“绝对不可能发生的”错误。然而，我们千万不要因此就轻易地根据这些偶然而片面的经历，得出某种错误的结论，认为——只要价格合适——在一个半年之内，上述的种种幸运之事，都会同时降临到自己团队的头上。

但是反过来，如果企图对各种可能出现的问题都在事先面面俱到地做好准备，我们就不得不需要在团队中设置很多附属人员。另外，如果整个任务必须通过划分再分配给一批人，那么随着任务规模的增加，需要进行的协调工作量也会同时增加。由于需要有部分精力花费在协调方面，所以一个大致的规律是：由3名程序员组成的团队，只能够完成1名能力相当的程序员所完成之工作量的2倍。另外，如果每个开发组分别由3名程序员组成，那么基于同样的原因，3个这样的开发组协作完成的工作量，将是单个开发组的2倍，或者说是单个程序员所能够完成的工作量的4倍。因此，假设某个工程由单个程序员需要8个月才能完成，如果我们希望在4个月内得到结果，那么我们就需要派上3名程序员；而如果我们希望在2个月内完成工作，就必须分配出9名程序员。

需要注意的是，如果程序员的能力大体相当，那么很可能2个月就是完成该任务所需要的最少时间——因为如果考虑到为把程序员们组织、协调起来所花费的时间，我们就有充分的理由怀疑，在短于2个月的时间内，即使是多达9名程序员，他们能做得出**什么**真正有用的**东西**。如果我们的确需要更早地得到最终的程序，我们就必须雇用水平更高的程序员来完成任务。

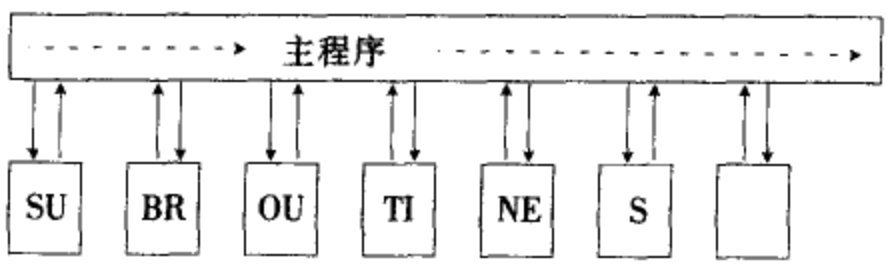
无论具体情况如何，程序开发团队的规模和组成似乎都符合这样一条基本的规律——如果希望通过最小的代价获得最佳的开发效果，你必须找到尽可能出色的程序员，并且给他们以尽可能长的时间，这样你需要的程序员数量也将最少。反之，如果你希望工作能尽可能快地完成，或者雇用

尽量少的经验丰富的程序员，那么开发成本与不确定性都会随之增加。当前最为常见的情况是：雇用一批实习生，然后给他们施加沉重的压力，同时又不提供必要的监督管理。无论如何，这种程序开发方式都必然是糟糕透顶的。

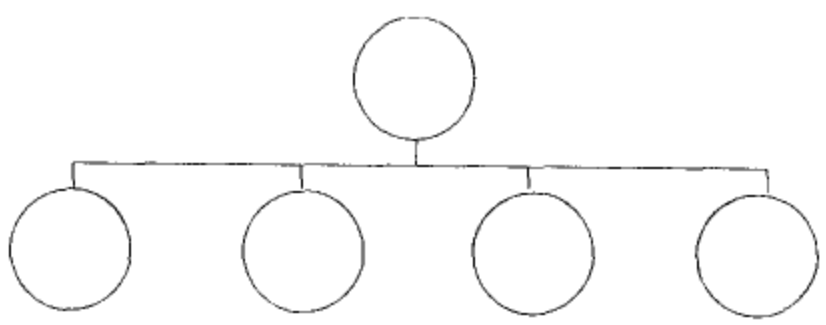
如果要求初出茅庐的实习生承担真刀真枪的任务，那么他们就无法有效地学习别人的程序开发经验。虽然有些团队的组织结构当前是最优的，**从长远来看**却不见得也是最优——因为要做到长期最优，我们还必须考虑完成培训的任务。所以，尽管我们可以预计到，那些（相对而言）缺乏经验的程序员对当前的任务贡献甚微，但是我们还是应该在开发团队中安排1～2名这样的程序员。一旦开发团队中有了这样的成员，团队的目标就变得更加多样化了——既有开发本身，也兼有培训——同时工作的组织方式也会受到影响。

就程序开发而言，开发团队针对其任务的组织形式，最主要地决定于以下两个因素——目标系统的组织结构以及团队的人员构成。实现一个系统结构的途径常常有很多种，但是能够投入工作的程序员开发组却很难超过一个，因此，开发团队的结构设计，通常需要与团队内各成员的长处与短处相适应。当然，最为理想的情况是，可以通过其他方式来选择一个团队的结构——首先规划出程序的理想结构，然后按照最优的方式，选取最合适的人选来承担对应的工作。然而，面对当前程序开发人才——甚至哪怕是“滥竽充数”——短缺的现实，此类计划只能是空中楼阁，无法在实际中得到兑现。

让我们来看看图5-1，这是一个关于系统结构与开发团队组织结构之间关系的例子，其中包括1名富有经验的程序员以及另外4名相对而言的实习生，该图说明了他们是如何构成一个系统的。a图表示一个程序是如何设置的。根据这种设置，它包括一个相当大的主程序，该主程序将通过调用相对更小——也许没有输入、输出——的子程序来实现其特定的功能。而通过b图可以看出，有经验的那位程序员可以担任该团队传统意义上的一位领导，由他自己来编写主程序，然后把各个子程序分配给实习生，而每个实习生所领受任务的规模与难度都是经过计算的，以保证与其能力的水平相对应。



a. 程序结构



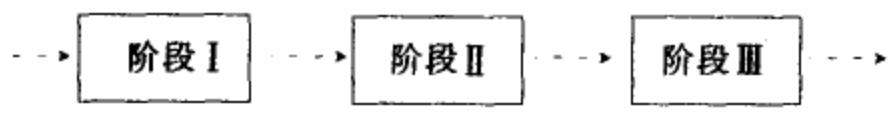
b. 团队结构

图 5-1　程序结构与团队结构（1）

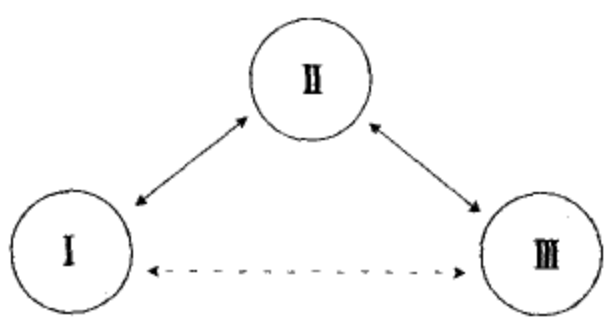
现在再来看看图 5-2，在一个由经验水平接近的 3 名程序员所组成的团队中，同样的任务将如何来组织。通过 a 图，我们看到现在整个程序是按照阶段，而不是子程序来组织的，各阶段环环相扣。当然，图 5-1 中的主程序也可以根据率领该团队的程序员的观念，按照阶段来进行组织，但是前一种版本中涉及界面的程序开发工作可能要比第二种版本更少。反过来，第二个开发组也可以在各阶段内部——甚至它们之间——使用子程序，但是在这种情况下，我们更愿意在最终的产品中看到尽量少的界面成分。程序开发团队自身的结构可能不仅仅是简单的对等个体之间的交流，正如图 5-2b 图所示的，由于成员Ⅱ的地位是介于其他两人的中间，他可能要花费更多的时间从事“联合性作业”。处于这样一种地位的成员Ⅱ，其承担的**程序开发性**工作实际上也许是最为轻松的。如果你注意到阶段Ⅰ与阶段Ⅲ已经分别对系统的输入、输出进行了处理，这一点就会看得更清楚。对于在整个系统中居于这个位置的他来说，与其他人密切合作的能力，也许要比程序开发的能力本身更为重要；正因为如此，也许作为一名程序员，他的能力是整个开发组中最差的。

当然，这些结构都是极其理想化的。比如，受到其所隶属的更大的开发组在程序开发方面的总体原则的影响，这些开发团队也会形成对应的内部结构，但是这里对此没有任何的反映。在实践中，如果推行无私式程序

开发，那么图5-1所示的组织结构可能不会包含那么多的层次；然而如果团队的成员在各方面参差不齐，那么图5-2中各部分的地位也许就不会那样的对等。在一个团队中，任何成员的地位都会在一系列因素的影响之下，不断地演化；其中，各人的工作分别由谁来评判，是最重要的因素之一。于是，如果采用了无私式程序开发，开发组中的每个成员，都有机会在某个时候检查其他的任何人的工作，因此这种方法会防止严格等级层次的建立。



a. 程序结构



b. 团队结构

图5-2　程序结构与团队结构（2）

在程序开发的过程中，开发团队中各成员的地位，通常在很大程度上取决于其他人对其个人能力的了解程度。如果程序能够（在团队中自由）流通，供程序员们相互评判、指正，那么“精英们”就能够更快地脱颖而出。反过来，如果团队的成员们都像苦行僧一样，隐居在各自的小隔间里独自工作，那么其在当前开发团队中的威望，就会更多地取决于其在此前建立起来的声誉——至少在大家意识到这种评价的错误之前的确如此，然而等到那时往往已经是无可救药了。

同样地，在被分配给某项特定工作之后，某个成员也可能相应地获得或者丧失一些地位。负责何种程序开发工作，将决定不同成员的身份地位——这就像办公室的级别决定了铺何种地毯一样。为了组织好一个团队，我们必须在推行工作的过程中，在各成员的敏感性格之间如履薄冰般地周旋。比如在某些情况下，编写一些诸如数据生成程序测试等附属程序

的人，也许是由于其工作并不会在最终的系统中实在地体现出来，所以这些人的地位将很低。尽管有些子程序需要克服重重困难才能完成，但是相对那些编写调用子程序的主程序的程序员而言，子程序的作者往往也居于从属的地位；或者是从子程序的从属地位类推过来的。不必经过**理性的**思考，这些拟人化的观点总会成为确定一支团队内部对任务分配满意程度的决定因素。这些因素的作用是实实在在的——或许正是因为其非理性的一面，它们的作用才更为实在。

目标的设定与认同

如果一个成员因为分配给他的工作而感到自卑，那么压抑这种感情将会给团队凝聚力带来灾难性的影响。采用无私式程序开发的方法，可以让每个人都感觉到自己并非偏于系统的一隅，而是共享了系统中的大部分，所以上述不良情绪将会得到缓解。虽然如此，如果一个团队的工作进展得过快——以至于在工程总体结构和任务分工等方面，还没有来得及在团队中建立起真正的共识——那么不是在这儿就是在那儿，问题迟早会出现。

在很多其他的文献中，社会心理学家们已经证实：只要有一名（或者多名）成员与集体的目标不一致，那么该集体的整体水平就将受到影响。这种影响不仅来自于这个成员本身，而且也来自于集体内部其他成员的绩效下降——因为他们总能感觉到集体内部的分歧或者某个同伴漠不关心的态度。按照图5-3所示的结构来组织的某个系统，就是一个非常离奇的例子。这个例子将说明：如果仅仅是在表面上达成集体的目标一致，那么必将付出多么巨大的代价。正如我们可以从图中的组织结构看到的，最初的开发团队包括7名成员，他们的能力水平不相上下；其中一人被指定负责编写输入/输出模块，为从一个阶段经入另一个阶段的数据提供一个统一的接口。

该系统在IBM704上运行了大约四年后，人们决定从704升级到709。从程序开发的角度来看，709与704几乎完全兼容——但是其输入/输出则截然不同。由于该系统的输入/输出功能被集成为一个单独的部分，所以人们预计这项转换工作会异常简单。在被移植到709系统后，输入/输出

子系统要求能够保持各个阶段之间原有的接口。在经过测试之后，新的子系统与原来的系统合并起来，所有方面都正常——但是到阶段Ⅴ时，系统却出现了问题。

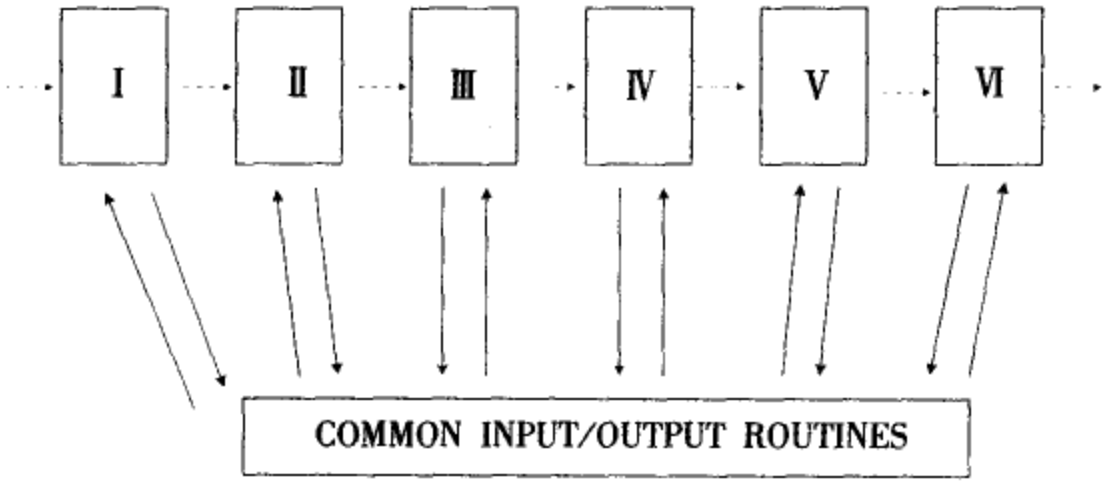


图 5-3　系统的组织结构

输入/输出子系统经过了反复检查，而且对阶段Ⅴ关于标准接口的所有变化可能都进行了检查，但是没有结果。于是，又对阶段Ⅳ进行检查，看看由它送往阶段Ⅴ的数据，到底有没有被新的系统破坏，然而结果显示：阶段Ⅳ的输出与此前完全相同，丝毫不差。人们还对猜测的其他所有方面进行了检查，但是经过了好几个星期，还是没有任何线索能够显示问题到底出在哪里。当然，在此期间压力也与日俱增，有人要求撤除 704，不再为两个系统同时支付租金。

系统的最初作者都没有在现场，但最后还是把其中一位邀请了回来，人们期望他能够为解决问题提供一点线索。他推测了一整天，同样也没有得到任何结果。于是他只好坐下来，和一位新程序员一边喝啤酒，一边回忆起他和当初那些伙计们在一起的美好时光。他忽然回忆起，Joe R. 曾经一度很烦恼，原因是他没有被委以编写输入/输出子系统的重任，而他认为没有人能够比自己更加胜任这项工作。

“大约有一个星期，Joe 都在为此而苦恼，”这位老前辈回忆到，“他感到自己的自尊受到了伤害，但是后来似乎他终于从这种苦恼中解脱了出来。他负责的工作虽然很琐碎，但是他完成得同样出色。”

“那他负责的是什么工作呢?”年轻的程序员一边询问，心中却隐约闪过一丝猜测。

“如果我没有记错的话，应该是第Ⅱ部分。但是你为什么问这呢?”

年轻人解释说，他只是有一种来自直觉的预感，然后就匆匆道别，径直朝计算中心奔去。他先找到一份第Ⅱ部分的程序清单——此前没有人阅读过它，因为人们的注意力一直放在第Ⅳ和第Ⅴ部分以及输入/输出子系统上——然后逐字逐句地进行分析。实际上他并不需要检查很多地方。就在程序的开头部分，该程序就从系统为输入/输出子系统保留的存储区中读取了一个字的数据，而在系统运行的整个过程中，这都属于核心的部分。然后，阶段Ⅱ又将该字替换为一个分支，而该分支则是跳往该程序员自己的一段代码——输入/输出子系统的流程就这样被重定向了。经过这些操作之后，该字原来的备份又被写回到原来的地方，于是所有的数据与最初进入系统时一模一样。

事情的经过显然是这样的：Joe最后根本没有接受开发团队的集体决定，只不过是不再公开地反对罢了；所以，他根据自己所负责的那个阶段的需要，擅自编写了单独的若干输入/输出函数。他也没有向任何其他人提及此事，而仅仅是在他那部分程序运行时，对部分标准的子程序做了短路处理。但是，等到新的输入/输出系统完成后，此前被该分支占据的部分将被用来存储一些变量，用来记录一台中介磁带的位置。因为在执行阶段Ⅱ程序的过程中这些变量被改变了，而在该阶段的最后又被恢复回了原始值，所以输入/输出子系统就无法继续对磁带的位置进行跟踪。而该磁带上的数据要到阶段Ⅴ才会被用上，所以看起来似乎问题是出在阶段Ⅴ上。这种由于成员认识不一致而造成的代价，要经过漫长的过程之后才能偿还；但是一旦时机成熟，就不得不偿还，而且会加倍地偿还。

在这里我们需要强调的是：我们这里所讨论的意见一致问题，只与集体的目标有关，而并非那些可以通过其他方式解决的问题。比如，在为银行开发一个分时系统的某个团队中，曾经发生过这样一件事：在对这个双向系统进行最后的验收测试时，人们突然发现这两台在线的打印机都会在正常输出的中间，打印出一行毫无用途的垃圾信息。这两行乱码完全一致，但是这两个子系统的运行完全相互独立——这样，就排除了由于一个系统的问题而导致另一个系统出现问题的可能。当然，两个系统采用的是同一个程序，因此所有的迹象都表明，应该是程序中存在错误。即使连电

源都是各自独立的，两台独立的计算机还是发生同样的错误、同时打印出相同的乱码行，而且这种事件发生的概率很难确定。

查找程序错误的任务，交给了开发团队中的一名队员。但是对证据研究了一个星期之后，他还是在原地没有任何的进展。第二个星期，另一位伙伴加入了进来，但是同样毫无结果。这时离程序交付使用只剩下最后的一个星期，鉴于该系统在其他所有方面都工作正常，为了讨论对这个难以捉摸的错误应该如何处理，专门召开了一个会。会场上，团队中的 11 名成员讨论得异常热烈，虽然未能获得任何结论，但是大部分人认为，应该把这个偶然事件看作是一个怪异的问题——一个双机错误！无论其他人再怎样地呼吁对该情况进行统计核查，也不能改变整个团队的意见——因为尽管它在自然上是荒谬的，但是在心理上它却是可以令人接受的。他们期望整个系统能够按时交付使用；在第二次会议上，剩下的人终于被说服了，该系统的开发工作就此结束。

该系统确实按照原来计划的日程投入使用，并且在头一个星期一切正常。但是，就在某一银行工作日的中间，正当需要使用该系统进行操作时，它却死机了——而且是两台机器同时死机。随后银行完全陷入了瘫痪，在经过约一个小时后，系统又重新启动了。没有人敢于估计在此过程中造成了多少业务损失，但是仅仅是事后对重新启动的后果进行的重新审计，就是一项耗时长达数月的浩大工程。在此期间，为了避免再次发生此类事件，整个开发团体都在疯狂地查找问题的根源所在。

不出所料，当这个错误最终被发现时，人们才知道，它就是早先造成打印乱码行的同一罪魁祸首——这是计时方面的一类偶然事件，其发生的概率为一个星期左右一次。在这种情况下，该偶然发生的错误还不至于对系统造成本质上的破坏——这也许是幸运的，也许又恰恰是不幸的——但是如果它再次爆发，情况就会更加糟糕。

这个例子所给我们的教训，属于意见一致的反面教训——实际上，这次的意见一致虽然不像上一例子那样属于表面上的一致，而是真正的一致，只不过是在错误意见上的一致。在程序开发过程中，即使不考虑此类问题，情况已经够糟糕了，但这的确是一个我们必须面对的心理学问题。在“观念及社会压力”这一题目下，我们对此还将有更深入的讨论。就目

前而言，主要需要汲取的教训是，应该注意这类意见一致与“集体目标的意见一致”之间的区别：健康的不同意见，将有助于对开发工作进行不带偏见的评估，但是前一类的意见一致将会扼杀这些不同意见，而后一类意见一致却会使得开发过程进行得更顺利。

为了实现在集体目标上真正的意见一致，最好的办法莫过于让开发组自己来确定其目标。首先，在目标的制定时的广泛参与，可以确保目标能够被大家充分理解。另外，这个过程使集体中的每个成员可以有机会对共同的目标做出公开的承诺，而——也许是因为认知失调的作用——这种公开的承诺已被证明可以提高目标的可接受性。这种参与和其他的因素不相干，但是参与本身却似乎是一项重要的因素，它能够决定各个成员是否真正地接受了开发团队的任务目标，并进而能够提高开发效率。

固然，并非每个程序开发团队都可以在任何时候、根据自己的偏好来确定任何的任务目标。通常，一个开发团队都是为了完成某个特定的任务而建立起来的，因此在团队建立之前，任务目标往往就已经确定了。但是程序开发的任务通常不会一下子就定义得足够明确，一般要等到开发团队组建起来之后，才能明确地落实，因此除非上层领导们非常专制，通常只要让团队中的所有成员一起行动，就可以确定意义明确的、有意思的目标。而最大的危险则来自于主管，由于他们都是经过程序开发中的层层选拔才得以出头的，所以这些人总是希望在手下成员们看到问题之前，就已经把每一个细枝末节都详细地定义好了。再没有其他的做法能够更加挫伤成员们的积极性了，它只能使成员们觉得自己不过是“编码器”而已。

如果一个开发团队是根据这样一种“逐字逐句的”定义开展工作的，就会导致其他的一些问题，而这仅仅是因为该集体对将要完成的任务不甚明确。“精确”和“清晰”的含义并不完全一样。目标的清晰程度，是影响目标可接受性的最关键因素；而为了实现“目标清晰”，就必须说明任务的具体含义，然后在这一含义所确定的框架下来对任务进行描述。程序员需要了解**“为什么做”**，而不仅仅是**“做什么”**。

如果开发团队的任务不是开发一个特定的系统，而是为其他的程序开发组提供服务和支持，那么这种目标不明确的问题就会显得更为突出。比如开发一个软件包，即使你根本不理解为什么要开发，但是至少你对要做

什么应该有个概念。但是提供支持的团队则不同，必须经常提醒他们注意自己的贡献，否则，他们的工作就会越来越具体，而且没有什么产出。

例如，在许多计算中心里都设有一支系统开发团队。这样一支团队的任务可能包罗万象，这些任务都围绕着为那里的其他用户提供服务这一主线。仅仅是由于对其所从事的工作缺乏明确而一致的理解，这样的团队也会蜕变成一些非常有趣、但是毫无用途的小型项目——我们知道这种情况的出现多么频繁。由于其他用户在系统问题方面缺乏能力，他们感到无法对系统程序员正在进行的工作进行评估，所以这种趋势就显得更为严重了。有这样一个不算特殊的例子：效力于同一所计算中心的三名系统程序员，曾一度沉迷于开发一个改进的装载器——这种装载器可能在有些环境中有用，但是在这个计算中心绝对没有任何用处。而对于来自其他用户的服务请求（比如，生成一个新的分类过程，或者把某个系统连接到图书馆等），他们都一概搪塞以“在现有系统内暂时无法实现”。事实上，他们甚至声称：所有这些要求都是可以做到的，不过得要等到新的装载器完成后才有可能。

他们所说的话本身确实不假，可以设想一下，既然整个团队都在忙着开发新的装载器，怎么会找得到什么人有精力来从事其他的工作呢？但是事实上，新的装载器与这些要求毫无关系，其惟一的作用就是消耗掉本来可以产生更多利润的劳动力外。因为用户们缺乏系统方面的知识（其中的大多数根本不明白装载器到底是怎么回事），所以他们只能受制于人、不敢申诉。最后，有一位勇敢的用户向外界的一位专家请求帮助，帮助他完成一项小系统任务。直到这时，事情才败露出来。虽然新的装载器根本就无法运行，但是他们却把系统的一次升级作为自己的借口；他们宣称，自己开发的装载器之所以无法运行，是因为系统更新的缘故——这样，他们的面子才得以保留。

除非计算中心拥有对团队的评估能力，可以判断他们是否正在按照指令开展工作，否则这种偏离团队目标的现象就无可救药。如果一支团队同时具有多个目标，那么即使每个目标都很明确，也会落入艰难的处境。除了此前曾经提到的存在于开发与培训之间的潜在矛盾之外，还有一些其他的矛盾——比如当一支开发团队需要承担两个程序的开发工作时。应该通过某种方式，在事先对两个工程相对的重要性加以明确，并且必须再三地

加以澄清，否则必然会出现偏向这个或者那个任务的情况。

当然，最好的情况是，一支开发团队只有一项明确定义的任务。然而不幸的是，这个世界上的事情很少会这样简单。即使其任务只是完成一个程序或者子程序，也有可能在诸如速度、空间或者时间进度等方面存在冲突，人们对这些方面的重视程度可能不同。如果没有明确地告诉每个成员应该对各个方面强调到何种地步，那么他们各自的工作目标就很有可能相互矛盾，其效果将互相抵消。可能有一位成员正在把大部分精力放在节省时间上，但是偏偏发现另一位伙伴正在费尽心思地把程序占用的空间压缩到最小——就这样，节约下来的时间就被浪费掉了。当这种矛盾暴露出来之后，就肯定会导致严重的冲突；特别是如果有关各方面在各自的工作中已经投入了相当多的精力，那么这种激烈的冲突就在所难免。

令人奇怪的是，如果不同成员对团队的目标理解不一致，而且这些不同观点之间的差异几乎无法察觉时，这些分歧的消除就显得极为迫切。人们常常会争论，在使用一个子程序时到底应该使用两个入口，还是使用一个入口外带一个附加参数。为了平息关于“到底应该从哪一头敲破鸡蛋”的争执，Gulliver 曾经伤透脑筋。但是与白白消耗在前面那些问题上的精力相比，Gulliver 消耗的气力也就算不得什么了[①]。如果在开发团队中持续出现这种争论，那么这就是一个确定的信号，它显示出在团队内部存在着更深层次的冲突——比如说，可能是为了争夺团队的“领导权”——所以绝对不能因为其貌似琐碎，就掉以轻心。

团队的领导者及其领导方法

“领导”这个词会令许多人联想起这样一种形象：希特勒（Adolph Hitler）只需轻轻努努他的小胡子，就可以让数百万的士兵踏着正步，迈

① 见 Jonathan Swift 的《格里佛游记》（Gulliver's Travels，1726），第一部的第Ⅵ和第Ⅴ章。梦想周游世界的 Gulliver 遭遇海难，流落到 Lilliput 帝国（小人国）。经过一段时间，Gulliver 逐渐获得了皇帝的信任，后来还因为帮助该帝国战胜入侵的 Blefuscu 帝国而被授予最高头衔 NARDAC。在此之前，这两个有着 6000 个月“悠久”历史的帝国，因为一个“天大的”问题——敲破鸡蛋的位置——已经进行了长达 630 个月的战争。作者在此引用这则故事是要说明，对一些“鸡毛蒜皮的”纠纷与争执，既不能神经过敏、反应过度，亦不可等闲视之。——译者注

向死亡的深渊。对于他下面的千百万民众而言，希特勒确实曾经是“至高无上的元首”，或者领导。但是类似希特勒这样的情况，更应该被理解为“盲从”，而不是“领导”——因为确实有某些类型的领导，只不过是众人对于某个特定个人人格的、根深蒂固的崇拜而已。

这种类型的领导，绝对不是我们的程序开发所需要的——通常，这也是难以令人忍受的。可以设想在一个典型的开发团队中，如果其主管手提长鞭高高在上，就像一位正在率领其分队进攻敌人机枪阵地的中尉一样发号施令，那么团队的成员会如何做出反应。不必去想像那些更为温和一点的独裁式“领导”，因为有许多无知的执行官，正在使用这种方法对待手下的程序员。一种典型的反应就是“消极怠工”。有这样一个消极怠工的例子，曾发生在某个项目中：当时正值项目的中途，其主管被更换了。新上任的主管看到有的程序员居然直到上午 10:30 才来上班，感到非常惊讶——而之所以如此大惊小怪，是因为他不知道，前一天晚上这些程序员可能在计算机上一直工作到了凌晨 2:00。

于是他发布了一条指示，要求严格遵守工作时间，员工必须依照公司制定的官方规定，被强制在上下班时打卡计时。程序员们的反应非常简单，无非是**严格**按照工作时间正常地上下班——一到下午5:50，整个办公室一瞬间就会变得空空如也；而在打卡计时器前，则常常会排起等待打卡的长龙。鉴于机时的短缺，该项目主要是采用交替换班的方式，因此这样马上就使生产效率降低了一半。在白天，程序员在等待程序运行结果的期间，只能闲着没事儿干，而每天的轮换结束后，计算机将立即关闭。

还有一个类似的故事，有位主管有一天发现，办公用品的消耗量超出了预算。于是他立即命令在办公用品仓库的大门上加上一把锁，只有负责办公用品的职员才有一把钥匙。程序员们的反应是，每个人每次都是单独来领取办公用品，于是在整个白天里，负责发放办公用品的职员都在疲于应付这些业务，根本没有时间去从事别的工作。最后，主管不得不再颁布一条命令，规定只能在上午 10:00～10:30 或者下午 3:00～3:30 之间领取办公用品。这种状况延续了一段时间，直到有一天，主管遇到一位程序员正盯着一面墙出神，于是便问他在做什么。



“我的钢笔用完了，”程序员回答道，“所以我只好等到明天上午10:00领到新的钢笔后，才能继续编写代码。”

主管的脸涨红了，谁都看得出来他生气了，“难道，你不可以使用铅笔吗?”他命令式地质问道。

“我一直都是用钢笔。你知道，用铅笔书写的程序，卡片穿孔员是不会接收的。那上面总是污迹斑斑的。”

“那么，你不会去借一支钢笔吗?”

“要是那样的话，我们就无法对办公用品的使用做精确的跟踪了，不是吗?”

不用说，在该主管被提升到管理层的更高位置之前，这个开发团队没有能开发出什么像样的东西。当然，他本可以用不服从领导或者其他的此类军事化的罪名来解雇这名程序员，但是他毕竟非常清楚，遇到一名糟糕的主管要比找到一名出色的程序员容易得多。对于程序开发的领导权力而言，程序员们惯有的缺点反而导致了整个领域中的一种有趣的现象。

至于哪些因素才是导致集体中的成员对工作感到满意的关键，社会学家在经过研究之后，划分出了四个主要方面：

1. 物质的奖励与机会；
2. 工作本身所具有的挑战性及其趣味性；
3. 其所隶属的更大团队的总体条件，比如雇员的福利、工作条件以及该团队在同类团队中的相对地位；
4. 主管与领导者的能力。

就程序开发的现状而言，头三条愿望已经可以通过规章得到满足，因此在决定程序员对工作的满意程度方面，它们一般的作用并不重要。可以肯定的是，在总体收入较高的这个行当之中，确实有一些低收入的例子；有些工作也确实枯燥乏味；也有些公司很糟糕。但是这些都不过是偶然出现的例外，而糟糕的管理以及领导，却要比我们所想像的更为普遍。这样，程序员们对待其“主管们”的态度，在很大程度上会导致其对工作的不满意——随之而来的必然是开发效率的降低——其程度也许超过了其他三个方面的总和。

就社会科学家对“领导能力”一词的使用而言，其含义应该是“对他

人的影响能力”。程序员们一般会更加注重创造性的工作以及专业能力，在他们所从事的工作范围内，他们会对那些被自己认定为很出色的人更加器重。这样，与那些世界级的口若悬河的推销商们相比，作为一名语调温和的程序开发奇才，将可以更加轻松地对程序员们进行领导（或影响）。当一名根本没有编过程序的领导者被指定负责某支开发团队时，除非该领导者明确地或者含蓄地承认自己在技术问题上的能力欠缺，否则注定会有麻烦发生。即便是那些曾经当过程序员的人，只要他后来有相当长的时间脱离了这个行当，那么如果他依然企图去与团队中的其他成员在程序开发才能方面一比高低，那么情况简直就糟糕透顶了。对于那些公开承认自己在程序开发方面经验欠缺的人，他的团队成员还有可能会尊重他；但是如果有人企图掩盖自己在这些方面的无知，那么等到他迟早露馅的时候，就会招致程序员们的无情嘲弄。

有一次，隶属于一个大项目的一支开发团队的领导者被提升后，Arnold C. 被指定去接替他。Arnold 的资历主要来自于其在数据处理方面的销售经历，然而他却假装自己在程序开发方面富有经验。在项目遇到一个关键的问题时，他却做得太过头了，居然“提供”了一个解决问题的“算法”。但是对于他手下极具经验的两名程序员来说，他显然根本没有弄明白自己在说什么。当他私下里向这两个人解释他的“方案”时，他们并没有立即指出其中的错误，相反地，他们却怂恿 Arnold 把项目中主要的程序员们召集起来，听他讲解他的想法。在会议中，他们在短短 5 分钟内，就让 Arnold 彻底地陷入了自相矛盾的境地。接下来，他们对他的“算法”进行了详细的解剖——详细到犹如在一个大项目中一样。最后，Arnold 在被召集来的人们的阵阵嘲笑声中离开了黑板；两个月后，应他的请求，他被调往另一个岗位。

由于这些领导者都是通过外部力量而任命的，他们并不具备为了在开发团队中实施自己的影响所必需的能力，因此，上述事例总是频频发生。这类“正式的”或者说“官方的”领导者所代表的只是来自团队外部的任务目标，所以其地位总是不稳定的；特别是如果他仅仅把自己的任务当作是“销售”，或者把“管理者的观点”强加给开发团队，那么其地位就会岌岌可危了。因为依照这种观点，管理方面根本没有妥协可言，所以一旦

在管理层与团队之间发生这类冲突，那么惟一的解决办法就是要求团队“妥协”。

即使假设团队会做出“妥协”，对于这样一个做过妥协的集体，只要认识到它的成绩，也没有人会期望看到这样的结果。但是强制的劳动——从最低的程度而言这仍然可以称为“劳动”的话——只能对相对不够成熟的程序员才可行，因为他们不了解还有其他的可能；或者对技术相对粗劣的程序员也可行，因为他们再清楚不过，其他的可能并不是为他们而存在的。然而在另一方面，普通的程序员既能意识到其他机会的存在，也懂得如何去抓住并利用它们。在极端的情况下，对这些方面了解过多，只能导致他们无休止地不断跳槽，但是却做不出什么成就。但是在正常情况下，与许多其他领域中的团队相比，对这些方面的了解会使程序开发团队以一种更加民主化的形式运转。

在一个民主化的集体中，领导能力——或者影响力——不仅限于某个人，一旦他的需求与其能力和思想相容，这种影响力就会在团队中从一个成员传递给另一个成员。但是，不同的人并非生来都是一样的——尽管他们生来就是平等的。所以，在一个民主化团队中的领导权力并不是均匀地向各个方向传播的，有些人所得到的会比其他人更多。民主化集体得以运转的一个重要前提，并不在于所有成员掌握同等的领导权力，而是在团队内部真实的集体生活之上（而不是来自外界的强加），确定领导权力的分配。所以，如果一个团队目前需要的是学习某些东西，那么团队中对相应的主题最为了解的那些成员，就会被提升到领导的位置，作为其他人的老师。另外，当团队的任务进入到调试阶段时，那些特别擅长调试的人，也会自然地成为经常性的领导者。

一支真正的民主化团队，总是特别能适应环境的改变，所以面对难以预料的困难，这种团队很容易成为一个值得信赖的开发团队。但是，主管们不愿意相信：一支民主化的团队会擅长于解决危机。作为证据，他们会列举出在某个领导的带动之下，一支集中式组织的团队所能实现的速度和判断力。他们讲的确实不错——如果这位领导碰巧是这种情况下的最佳人选的话。比如通过这样的命令方式，希特勒也的确凭借其能力获得了巨大的成功——但是它的对手们也是按照同样的形式组织起来的。也许，在希

特勒失败之前，他的确是那个时代的合适人选。当然，只要有足够多的人才和智力可以利用，那么在一支按照民主的方式组织起来的团队中，合适的人选每次都能自然地被挑选出来。比如在瑞士，在面临战争的威胁时，就会选举出一位将军来指挥其军队。而当战争的威胁消除之后，将军也将随之取消——但是，任何时候都会根据当时对领导权力的需要，选拔出相应的领导者。与某些国家不同，瑞士绝不会在和平时期因为某个将军妄图统治整个国家而付出代价。

不幸的是，就可以在任何时候自由地挑选最佳人选担当领导者这一意义而言，没有哪个程序开发团队可以做到百分之百的民主化。一方面，团队内部未必有这样的最佳人选。其次，人格的复杂因素可能会影响到对当前最佳人选的选拔。最后，被外部指定的领导者的介入，本身就往往会使得团队偏离民主化的方向。

来自外部的力量总希望去影响团队自身追求的目标，在这二者之间，被指派的领导者实际上是一名信息的携带者。在理想的情况下，他可能是一位中立的信使，但是这种理想情况从未出现过。与外部团队的交往，使得他可以使用一些特殊的信息来源，而团队中的其他人却无法共享这些信息渠道；这样，他就可以利用这些渠道作为手段，把自己的领导权力提高到更强的地步。比如，如果管理层告诉他说，他们希望某个程序能够在 10 个星期内完成，他就会告诉手下的成员说，管理层希望的期限是 8 个星期。这样，如果团队成员希望的期限是 12 个星期，他就可以造成一种假象，使人们认为是他为了尊重大家的意见，成功地与上层领导达成了一个妥协，将期限改为 10 个星期。对于他编造的这种“帮助”大家的假象，其他人根本不可能知道那只不过是他利用其特殊地位而玩弄的一个伎俩。尽管他不过只是一名“管理层的奴仆”，但是作为领导者，他的威望却会因此得到了提高。

当然，一旦团队的成员们发现，他仅仅是因为个人的私念，歪曲篡改了来自管理层的信息，那么他就不可能继续担当该团队的一名有效的领导者。然而，对于这种被指派的领导者，由于他们的地位（相对于程序员）的非对称性，那种来自不诚实和玩弄手腕的诱惑非常之大。尽管他主要是与团队成员们一道工作，但是无论是对他的奖励还是惩罚，都来自更高级

别的管理层。通常，在一名程序员同意接受担当团队领导者的指派之时，他就已经对跻身管理层盼望已久了。在这个方向上的进一步提升，已经与其程序开发能力——尽管正是凭借这种能力，他才到达今天的地位——毫无关系，而他今后的升迁将取决于他取悦于管理层的能力。

目光短浅、难以依靠的团队领导者可能会认为，博得管理层欢心的最好方法，就是无论他们要求什么都满口答应。但是最后，管理层需要的不只是诺言，而更重要的是**恪守**诺言；而只有在团队的领导者有能力使整个团队都接受这个诺言，并且以此作为集体的目标，诺言才有可能兑现。团队的领导者们需要学习的东西包括：

1. 无论主管们怎样地强调诺言，他们真正关心的只是结果；

2. 如果希望得到的结果与在整个团队的参与下所确定的工作目标一致，那么这一目标就会非常轻易地实现。

只要牢记这些事实，被指派的团队领导者就可能克服由于被指派而造成的困难，进而成为一位独立自主的领导者。他也会把“指手画脚、发号施令的能力”转变成“领导能力”。

但是，主管并非总是清楚他们需要的是结果，如果不能承认这一点，我们就不是真正的诚实。我们经常看到的一种情况是，因为其队员确信某项任务无法实现，团队领导者因而拒绝对上层做出相应的承诺，所以他被调离其岗位。这类人员的撤换会一直持续下去，直到出现某位候选人，他渴望的是自己的提升，而不是可以真正令他抓住成功机会的那些有才能的头脑。很不幸，如果是一项长期的项目，那么这位候选人就会因为他对上层领导的唯命是从而步步高升，直到该项目终于到了就连那些高层领导也能看出他的承诺根本无法兑现的境地。这也是一种类似“Musical Chairs”[①] 的游戏，游戏的目的就是，在外界觉悟到项目注定要失败的时候，不要处在承担责任的位置。这种游戏玩了一次又一次，其效果是把不止一位程序开发主管送上了他们今天的高官显位。

然而，作为一名被指派的团队领导者，如果他的手下无法完成某项任务，但是他的上级却强迫他接受这项任务，那么他所能采取的最好方

① 一种抢座位游戏。——译者注

法，就是坚决予以抵制。自然地，他必须确保还能接收到新的信息，这些信息可能改变他的处境——比如当上级管理层对他承诺，将给他分配更多的人力和更优惠的上机时间。但是，他必须为自己评估一下，到底这些承诺兑现的可能性有多大；因为一旦**他**向自己的上级做出了承诺，那么他们就很可能把他从他们那里争取到的条件扔到九霄云外。但是当管理层转而讨论除资源或者任务定义之外的其他方面时，他的判断绝对不能有丝毫的动摇。最为典型的承诺是，在事成之后将给予他如何如何特殊的奖励，但是他必须保持头脑清醒，千万不要被奖励的诱惑冲昏了头脑，而不考虑得到它的可能性到底有多大。

当上级管理层提出奖励的许诺时，团队领导者除了一味地抵制之外，别无良方。他必须乐于借助自己的专业判断力量，来支撑自己作为一位被指派的领导者的地位。如果他本身是位优秀的程序员，那么他在这场斗争中就拥有双重的砝码——因为他对自己的判断更加充满自信，同时他也很清楚，即使他丢失了这份领导者的职位，也不至于沦落到讨饭的地步。然而，要是他对自己的专业精通程度有所怀疑，那么面对这种暗藏威胁的、微不足道的承诺，他很容易就会上当受骗。

为了说明本节的要点，让我们来看看 Harold M. 的例子。Harold 是一位出众的程序员，但是有一次他却发现自己成为了一名受夹板气的被指派领导者。他先前曾接受了一项任务，带领一支团队开发一个编译器。本来任务完成的期限已经确定，但是后来管理层又希望把任务的范围进行扩展，但是日程安排照旧。在与自己的团队讨论之后，Harold 与手下人的意见一致：除非开发期限延长三个月，否则他们无法按时完成任务。当 Harold 把这个意见返回给他的上级时，他才发现这个意见不会被接受。当他坚持自己的意见时，他的主管召集了一个会议，参加者有 Harold 、Harold 的主管以及主管的主管。

根据参加会议的人员组成，Harold 敏锐地觉察出他们正企图采用“恩威并施”的方法，令其就范。他们的安排非常巧妙，采用了一种非常轻松的私下会谈的方式。这些人努力营造出一种气氛，这种气氛可以让 Harold 体会到自己的重要性。在这样的气氛中，这些人采用威胁和承诺双管齐下的策略，其手法之巧妙令常人难以觉察，因此很容易就会麻痹大意、放松

警惕。但是 Harold 已经经历过太多的这类场面，所以他可以预测出每一步动作，并且已经用对自己及其团队判断的信心来武装自己。在会议进行了一段时间、周围人的进攻每次都被巧妙地回避过去后，会场的气氛开始有所升温。最后，会议进入了终极的威胁阶段。

“Harold，”他的顶头主管启动了话题——实际上，该主管的任务就是唱白脸；而主管的主管的任务，则是扮演一位仁慈父亲的角色，把主管刺耳的威胁柔化为甜蜜的承诺。“Harold，”他重新开始说道，因为刚才他停顿了一下，把自己的椅子往前面挪了一下，移动了足足三英寸，“你没有与我们很好地配合。作为一名主管，你应该学会这一点；还有妥协，也许你并不具备作为一名主管的条件。”

Harold 领会出了其中的潜台词，但是他并未放过这一点，而是直截了当地把问题抖落开来：“我认为你可能是对的。既然如此，为什么你不另请高明呢？——应该有很多人懂得如何配合。”

主管的主管看到 Harold 已经推翻了他的限制，于是就试图把方向重新引回到正面的策略上来。“但是，Harold，”他奉承道，“没有其他人选。你是惟一知道应该如何开展这项工作的人。我们要依靠你。”

“既然如此，”Harold 说道，他的声音中夹带着嘲讽式的胜利感：“既然我是可以开展这项工作的惟一人选，那么在我告诉你们不可能在那么短的时间内完成任务时，你们为什么不相信我呢？”

“因为你并没有**真正**理解这个问题，”Harold 的主管又开始进攻了，但是 Harold 此时早已离开了座位，向大门走去。

“好吧！”在即将离开这次已经陷于死锁的会议之前，他说道。“你们需要选择一下。如果我的确是惟一理解这个问题的人，那么我就是惟一**真正**理解它的人，于是你们就必须按照我的方式来行事。否则，尽管替换我好了，不会有任何问题。这是你们的问题，与我没有关系。所以请原谅，我要回去工作了。”

Harold 失去了他的工作吗？这次并没有；但是真正重要的并不在此，因为他很清楚，即便那样，他也可以得到另外一份工作，而且不会比这份差。如果这份工作的确重要，他就不可能那样做了。领导能力方面的一个悖论非常简单：只有随时准备下台的领导者，才有可能获得成功。

团队中可能出现的危机

一支程序开发团队的生命周期，首先是从招募队员开始；接下来需要进行目标设定，以及根据任务建立组织；当最后任务完成时，团队将被解散，或者继续接受新的任务。但是，在这个过程中充满了这样或者那样的危机，以至于任何一个普通程序开发团队的故事，都可以写成一部十分有趣的——尽管有点难以置信——小说[①]。一种很有用的办法就是，把团队的工作划分为两种类型分别对待：以完成团队任务为直接目标的工作，以及在面临危机时以维护团队的有效运行为直接目标的工作。对于社会心理学家们来说，它们分别被称为“基于任务的”和“基于维护的”工作，关于这个课题，已经有了很多的研究。

在某些类型的集体中，通常是在开发团队中，成员们会倾向于推选两名互补的领导者：一位是任务权威，他负责设定和分配任务，并进行必要的协调；另一位则是协调工作的权威，当集体内部的成员之间、个人目标与集体目标之间发生冲突时，由她来负责调解纠纷。对开发团队内部来说，由上级指定的领导者的任务是传达和落实来自外部的指令，所以他们常常会担当任务权威的职位；当然，正如我们所知道的，如果这种人不能显示出必要的才能，就有可能被他所领导的集体替换掉。负责协调维护的权威则常常是集体中最讨人喜欢的人，这类人可以来自任何地方。在程序开发方面，虽然这种人可能还过得去，但是不一定是高手。一种常见的情况是，这个人是位女性。

在对核心家庭[②]进行的一项文化交叉研究中，至少从文化标准来看，通常也能发现类似的任务与协调功能的划分。虽然不是在所有的文化环境中，但是在包括我们在内的大多数文化环境中，人们理想中的父亲是一位任务权威，而理想中的母亲则是一位负责协调的权威。所以，通常由女性

① Tom DeMarco 1997 年曾出版过一本有关程序开发的畅销书《最后期限》（*The Deadline: A Novel about Project Management*），其中文翻译版已由清华大学出版社引进出版。这是中国第一本有关软件开发的通俗读物，书中用一个虚构的故事阐述了真实世界中关于软件开发的一般问题。故事中出现的温尼佩格即是 Weinberg 的化身。——编注

② Nuclear family：核心家庭，由父母与子女组成的家庭结构。——译者注

程序员来担当这个角色，也许在我们的文化环境中是很自然的。在推选协调权威时，性别是不是一个重要的因素呢？尽管还没有通过研究来证实这一点，但是无论如何，经常发生的一种情况是，只要在一支开发团队中有一位女性存在，她往往就会被推上“团队妈妈”的位置。我至少见过在好几个开发团队中，总有一位女性被公开地称呼为“团队妈妈”（team-mother）或者“小队妈妈”；（den-mother）此外，在计算业界的一个经久不衰的玩笑就是，把“软件”（software）戏称为“女性程序员”（girl programmer）。“柔软”这个词多被用以形容协调性的工作，而“坚硬”则意味着对工作进行指导和命令。

无论这种按照性别对领导工作进行的分工是否能够最终在社会心理学中找到依据，但是这方面大量的轶事都说明，在每支开发团队中安排至少一名女性成员，哪怕并不是作为指派的领导者，也是非常值得采取的一项措施。正如在许多家庭中所看到的情况一样，女性没有担当任务权威的角色——当然，为什么如此，至今还没有确定的解释。但是主管们应该特别小心，不要仅仅因为一位女性在团队中起到了副指挥的作用，就指定她担任领导者。实际上，对女性采取这种限制毫无必要；如果需要确定一支团队的协调权威，那么在进行提拔之前一定要三思而行，因为这种提拔的官方行为的特点，它可能会对这个角色的领导绩效产生负面影响。

在开发团队的生命期内，对某一位领导人或者任何团队成员进行更换，可能是司空见惯的家常便饭。而增加或调离某个成员所造成的影响，与该集体的内部结构密切相关；即使某个成员在团队中看起来很不起眼，但是在他调离或者加入该团队之后，很多主管都会吃惊而苦恼地发现，该团队的开发效率有很大变化。如果更高的管理层要求调离团队中的某个成员，其理由不外乎是因为此人能力出众。在这种情况下，如果他的调离影响到整个团队的工作绩效，那么当然没有什么大惊小怪的。但是，工作绩效方面的损失，并不总是可以完全归咎于某个天才的调离——这个道理，就像整个团队的绩效，并不是团队中每个成员各自工作的简单总和一样。

无论是因为提升、退休、生病、故去、怀孕，还是突然有一天看破红尘，去庙里出家做起了和尚，团队的成员都可能中途离队。在某些成员离开之后，“民主式”组织的开发团队显得更有能力经受住相应的影响。由

于任务是由全体成员共同承担的，同时他们之间存在很多交流，所以当某个人离开之后，其他的伙伴会根据需要填补他的空缺，并且运用在他们共同工作期间掌握到的知识——这样，此人离开后的空缺并不会引人注目。但是反过来事实也证明，如果有一位新来者加入到一个民主式管理的团队中，那么由于其内部结构中不会专门为他留出一个明确的位置，所以这样的团队通常很难接受新成员。从外界看来，一个民主式的团队的外表显得更加冷漠和不友好，而一个集权式的团队则会对外来的加入者表现出异乎寻常的热情和友好——这似乎是一个悖论。

在一个集权式团队中，如果到了所有的工作都必须通过领导者来部署和监督的地步，那么在任何成员离开后自然就会暴露出任务分工上的空缺。而为了在剩余的成员中重新进行任务分配，只有其领导者才拥有必要的信息。如果这位领导人正忙于其他事务或者外出了，那么任何一名成员的离开都将会给整个集体带来灾难性的后果。

但是反过来，当有新成员加入这种集体时，手续就显得更为简单，因为他要做的就是去找团队的领导人，由领导人来指定他负责“这项、这项或者那项”任务——其中各项任务大多都是从其他的某个伙伴那里接手过来的。在更换成员时，只要新成员的能力与老成员相当，手续也可能同样简单，因为需要做的无非是把原来那人的工作重新分配给接手的人。但是，如果仅仅因为离开者对项目的了解是通过此前的工作积累下来的，那么就这个意义而言，我们很少能够遇到那样纯粹的一对一的更换。在通常出现任务不匹配的情况时，民主式团队更有能力对工作进行重组，以符合当时的现实情况。

有的时候开发团队中的成员会突然意识到，其中的某个伙伴没有能力胜任其承担的那部分工作，这时就会发生另一类危机。在民主式团队中，处理这种危机的最可能的办法，就是逐渐地把该成员不能胜任的工作转移给其他成员。而如果该开发团队是由一位强硬的领导集中式管理的，那么最有可能的解决办法则是直接解除该成员的工作。但是，实际情况要远比这复杂——因为等到发现这个问题时，可能已经没有足够的时间来找到接替的人选，并对其进行培训，所以简单的解雇某个人可能于事无补。但是在一支民主式的团队中，因为其中的每个成员都有很多机会来审查其他伙伴的工作，所以如果存在这种问题，那么这一问题必然会很早就会被发

现。因此，只有当成员能力上的欠缺异常明显时，独裁式的管理办法才有可能更加有效。即便如此，因为即使是能力最差的人，也可能会深受同事的喜爱，所以简单生硬地把某个不称职的成员开除掉，不大可能会对集体的士气有什么鼓舞作用。

在一支民主式团队中，如果某个成员能力很强，但是却与周围的人合不来，那么这个问题就会比某个成员能力极差的情况更为严重。在一支集权式的团队中，这样的一个人根本不需要在工作方面与其他同事有多少接触，所以只要能够与领导人处得来，他就不会有任何特别的问题。的确有一些程序员甚至更愿意在一位态度强硬、唯我独尊的领导人手下工作，因为这样他们就无需与同事们进行任何社交活动。然而一位不善社交的成员会切断自己与他人的任何联系，所以在团队会议中他总是表现得结结巴巴，这将成为在一支民主式团队中达成意见一致的障碍。

开发团队有很多种办法可以使其中的成员加入到其社交圈中，这正如排斥社交的行为可能有多方面的原因一样。在智力上远远超出其他同事的成员，也可能会有此类排斥社交的倾向——因为他可能“见到怂人就压不住火”。但是另一方面，他周围的同事们也无法承认或实现他的那种建议。如果一个人不会运用他的智慧去调整其社交的行为方式以及与人交往的方式，那么他的聪明才智反而会给他的程序开发工作帮倒忙。

George G. 曾经作为一名实习生，被派往一个负责诊断程序的开发组，这是一项无论从哪个方向看都很重要的工作。他刚刚以优异的成绩，取得数学专业的硕士学位；但是在他的七名同事中，只有两人曾经上过大学。这个团队的任务是为军用计算机开发一个实时的诊断程序，而 George 则负责对高速磁鼓的诊断——这是整个系统中的关键部分。虽然这个程序是他的处女作，他却成功地设计了一个极具创造性的方案：程序首先把所有磁鼓置为“1”，然后通过对读回来的数据中“0”出现的模式，就可以准确地定位是哪条线路出了故障。

George 为此得到了极高的赞扬。可是他并不需要这些赞扬，因为他心里非常清楚，在周围同事们“缺乏想像力的”工作之中，他的工作简直就是鹤立鸡群。George 的确没有放过任何可以指出别人缺点的机会，这些缺点在他看来再明显不过，只需与他的程序稍加对比，这些人程序中的丑陋

之处就暴露无遗。没过多久，开发团队中就开始酝酿一次严重的危机：有几个人试图把 George 排挤出开发团队；而有些人则采取不同的方式，他们在外面为自己寻找新的职位。最后，命运终于介入了进来：一位军方的高官前来参观，并且顺便视察一下新硬件的研究进展。

尽管团队事先为这次视察进行了充分的准备，但是偏偏在客人到达时，计算机却突然功能失灵。诊断程序运行得非常成功，但是其他的软件却根本不起作用。诊断程序重新运行之后，虽然它本身照旧能够运行，但是常规程序却还是无法工作。看到事情弄成这个样子，军方人士开始有点不耐烦起来，而管理层的领导们则近乎狂怒了。终于，有人无意间踱到主机背后，发现有一条电缆接头松了。他不经意地问这根电缆是做什么用的，这时人们才发现问题正出在那里。原来，磁鼓根本没有被接到系统上！

磁鼓终于被连接好了，接下来的演示很精彩，加上午餐时的马提尼酒，军方人士终于感到了些许安慰。但是，直到这批人离开很久以后，团队中的其他程序员们还在借各种机会报复 George。当磁鼓没有连接的时候，他错误地忽略了与磁鼓之间的界面。根据系统定义，如果选中了某个尚未连接的设备，应该给出一个中断信号，但是 George 却屏蔽了这种中断。而一旦中断被屏蔽后，对磁鼓的写入操作将直接被忽略掉，而读取操作则将返回一串“1”。而虽然这时磁鼓根本还没有连入系统，George 那“绝妙的”程序也会“判断”出：磁鼓一切正常。

任何程序员都有可能犯这类愚蠢的错误，这种问题也很容易修补好。错误的原因并不是 George 的骄傲，而是其他的程序员在利用一切可能的机会来戏弄他。也许，要是他没有宣称自己的程序“绝对可靠”，这次大跟头就不会对他有如此大的伤害。但是落到这般田地，George 实在无法承受。两个星期之后的那个星期一，他没能在办公室露面，这也是人们最后一次听到关于他的消息。虽然这种结局让人感到不寒而栗，但是它却是解决问题的有效办法。事实上，整个团队在不久之后便恢复到了以往那种令人愉快的状态——人们不断地编写着“缺乏想像力的”诊断程序，但是，这些程序非常健壮。

在一支团队生命期内所发生的危机，不见得一定会与某个特定的人联系起来，而是可能在团队生命期中的任何阶段都会出现——可能作为或多

或少正常的“成长期”，或者是为了适应外部环境的改变。比如我们可能会估计到的：一支团队的形成是其生命期中的关键阶段——因为通过这个阶段，原来没有的组织结构现在形成了。按照一般的规律，团队的组织结构是同时沿自顶向下、从下而上两个方向建立起来的——这就像冰块在碟子里的凝结过程一样。然而这只是一个完全根据经验的观察，但我们可以估计出，真正的原因在于开发团队中超凡的能力很容易被发现，并进而从众人中脱颖而出。但是有的时候情况并非如此，比如有时在同一支团队中可能存在着两名潜在的强权式领导者，在这种情况下，自上而下的组建工作就可能会久拖不决。一般而言，这样下去将无法达到预期目的，但是只要正确地进行团队划分，这种问题就很容易避免。如果在某个项目中存在着许多潜在的强权式领导者，那么与只有少数领导者的情况下相比，就可以划分出更多的开发团队——为了避免在组织建设过程中的矛盾冲突，这种方法很有必要。

那么谁会是潜在的强权式领导者呢？正如我们所看到的，这取决于开发团队所承担的任务要求。在团队的生命期内，这些要求会不断调整，所以领导者的命运也会相应地起起伏伏。对于团队组织结构方面的这些调整，如果因为性格上的原因难以为人接受，那么一旦进入了一个新的工作阶段，人际冲突将必然发生。因此，还不能说在团队组建起来后，最多不会经过太久，其内部结构的设置就可以一劳永逸地大功告成。由于我们无法在事先预测，在各个阶段谁将是自然而然的领导者，所以在挑选团队成员时，最理想的人选应该是程序开发能力与人际交往能力俱佳的——不过，这很难做到。这样，我们必然总是会看到形形色色的冲突不断出现——其共同之处在于，每一次都会使开发效率降低。

不同任务阶段之间的变化，可能会更加平缓一些，这也可以平息那些由组织结构的变更而引发的冲突——尤其是当开发团队的成员之间交流充分时，解决这些冲突的小机会还是很多的；此时，各任务阶段之间变化的作用就显得尤为重要。不过在有些时候，受到来自外来的人或者事件的影响，这些变化可能来得太快。在开发团队的生命期内，引发危机的最常见事件包括：计算机功能失灵、计算机任务超载、关键环节上无法排除的程序错误、两个分别测试过的程序在进行系统联调时出现问题、开发计划的

更改、新设备的引入、高级管理层的人事变动，以及任务定义的变更。毫不奇怪，似乎在每个程序开发团队中，危机的存在总是家常便饭。

在社会心理学中，有两个关于集体行为的一般性观察结果，这些结果与饱受危机之苦的程序开发团队有特别的联系。首先，人们注意到，当危机发生时，相关集体中的各成员将更容易接受相对更为强权式的领导方式。然而与此同时，如果有人自诩为领导，但是其命令却无法尽快而有效地平息冲突，那么大家会渐渐失去耐心。于是在一支开发团队中，随着危机的此起彼伏，领导权的格局可能总是处于变化之中。得益于这种形式的改组，任务越是艰巨，所有成员反而会越发紧密地团结在能够在实际上更加有效地驾驭整个团队的领导人周围。

因此我们将明白，对于程序开发团队来说，为什么民主式的——也许应该称为“崇尚技术式的”——组织形式更为自然。这种自我调节式的结构既不会过于自主，同时也不会过于被动。在为开发团队挑选程序员时，我们应该尽量考虑选择更加适应这种结构的那些人。在对程序员进行培训时，我们应该尽量地教会他们如何团结在有才能的领导人周围；教会他们在自己成为了集体中最有资格的领导者时，如何抓住机遇，勇敢地挑起领导的重任。此外，虽然从表面上看，团队内部的民主化进程会带来一些损失，但是从长远来看，这将会保证开发团队更为有效地运转；所以如果我们身处局外，那么在开发团队的生命期内，我们还是应该尽可能地去避免对这些进程带来任何的干扰。

其实，一旦团队建立起来并开始运作，其上级主管要是聪明的话，就应该对其内部结构以及结构的调整采取一种“袖手旁观”的政策。与司空见惯的情况一样，在团队成员之间发生争论的时候，如果其中的某个成员来找他，并且希望从他这里获得倾向于自己的权威性意见，那么他就应该毫不犹豫地效仿老拉比[①]的方法。有一天拉比独自端坐，正在研究问题，一位情绪激动的男子闯了进来要求见他。这个人给他讲了一个长长的故

① rabbi，拉比，犹太人的学者，受尊敬的宗教知识分子。公元 70 年，犹太教的中心耶路撒冷圣殿被罗马兵夷为平地。犹太祭司、撒都该派和奋锐党人遭受沉重的打击，只有坚持信仰又不积极参与斗争的法利赛党从耶路撒冷转移到外地，建立研习律法、维持宗教生活的中心。此后，精通熟习摩西律法的法利赛派律法师逐渐在犹太社会里取得了领导的地位，被犹太群众尊称为 rabbi。在此后的六个多世纪里，犹太教也因此被称为拉比犹太教。——译者注

事——其实无非是关于他与妻子之间的一次小小的争论。在讲述完毕后，此人恳求拉比评判一下：到底是他对，还是他的妻子对。

“你是对的，”拉比说完后，那名男子就兴高采烈地作别而去。然而才过了不久，那人的妻子也来找拉比——当然，她显得比她丈夫还要激动。

“你是什么意思？”她质问道，“为什么你说我丈夫是对的？你不应该听信一面之词，现在就请让我来讲讲吧。”接下来，她又从自己的角度讲述了一遍事情的经过，最后，她要求拉比重新做一次评判。

“你是对的，”拉比的话刚说完，那人的妻子也心满意足地离开了。但是，这回该轮到拉比自己的妻子不满意了——因为她从头到尾听过了那夫妻俩的讲述，也听到了丈夫的回答。

“你怎么可以那样做？”她问自己的丈夫：“你告诉那位丈夫说他是对的，然后又对他的妻子说她也是对的。但是，他们俩绝不可能都对。”

“你是对的。”拉比说。

小结

在很多程序开发的环境中，最基本的工作单位就是开发团队，而不是个人。尽管可以从社会心理学家那里得到很多有关“小型集体的行为”的实验结果，但是除了那些非常有启发性的趣闻轶事外，目前我们尚没有对程序开发团队进行过真正的研究。当与来自社会心理学的结果联系起来后，这些趣闻轶事能够使我们强烈地感觉到，那些将影响到一支团队的生命期及其效绩的因素。这些因素包括：

1. 各成员的特长与不足；
2. 目标设定的方式；
3. 待开发的程序的结构；
4. 由外界强加的领导管理结构；
5. 某些成员的性别，以及其他成员对待这种性别的态度；
6. 团队与其周围环境中其他部分之间的沟通联系；
7. 团队领导人在技术方面的能力与欠缺。

通过将这些趣闻轶事的材料与实验情况进行对比我们发现，我们必须对其中的一项差别特别谨慎。在实际的程序开发环境中，其时间的跨度以及任务的复杂程度，都要比社会心理学家曾经试图进行的实验高出几个数量级。社会心理学实验中的任务只持续一个小时、一天或者至多一个星期。如果知道持续时间很短，人们就可能会压抑住自己的感情；反过来，他们也不会认为压抑自己的感情有什么不对劲儿，因为他们以后将不再会遇到这个临时性集体中的其他成员。通过对由陌生人组成的团队和由老同事组成的团队的比较，我们的确发现，与团队成员过去合作的经历，以及对将来能够和他们一道工作的期望，都会影响到短期集体的行为。因此即使从最乐观的角度来看，要想对开发团队进行任何有意义的试验，都不是一件轻而易举的事。

但是，即使是对其他领域里正在工作中的团队进行的观察，除非这些观察涉及到某项正在进行的异常复杂的、通常时间要求很紧、并且工作条件生疏的智力型工作，否则我们都不能放心地将这些观察结果直接应用于对程序开发的研究。在这里，我们可以利用在最近的研究中所取得的几个原始实验数据——其中作为研究对象的团队来自医院或者科研实验室。但是即使是在这里，依然缺乏一支程序开发团队所具备的某些要素——比如，犹如上帝一般矗立在那里、对我们的工作进行评判的那台无所不知的计算机。比如在医院里如果病人康复了，人们会认为这就是医生的功劳；而要是病人不幸死了，那人们就会认为这是上帝干的。可是，当计算机程序无法运行时，任何人都会知道——而且任何人都知道应该由谁对此负责。

思考题

请主管回答：

1. 为了使自己对整个团队的指挥更加统一，你是否会雇用实习生？在组建开发团队时，你是强调每个成员应该能够更好地“融于”集体之

中，还是采用相反的标准？

2. 对于在开发团队中安排女性成员，你是怎样看待的？又是如何做的？对于指定女性作为开发团队的领导者，你又会如何看待呢？你能否举出一些基于经验的证据，来支持你的看法和做法？

3. 在你所在的集体中，有多少决策是由开发团队中的所有成员共同做出的？主管们“逐字逐句的”工作有多少？你自己呢？

4. 你是否曾经在你手下人的面前，吹嘘过自己的专业能力？描述一下这样的一些故事。你是否曾经发现，自己至少在某个方面不如手下的某位程序员？描述一下这样的事例。这一发现的后果是什么？掩盖这些事实是否值得？

5. 在设定你自己的工作目标时，哪些是由上级领导层自上而下指定的？哪些又是根据下层程序员的意志自下而上确定的？你对这些安排是否满意？你是否会通过某些方式改变这种情况？

6. 你是否曾经将团队的“二把手”提升为团队的主管？如果是，请解释一下你这样做是为什么？并请你描述一下在这样做之后，出现过什么效果？

7. 在团队内部出现争执时，你是否曾经被要求做过评判？如果是，描述一下当时的情况——你是如何做的，以及结果如何？

请程序员回答：

1. 你最拿手的是程序开发中的哪部分工作？你在团队中所担任的工作，是否与你的这一长处吻合？大家是否普遍认可你最擅长于这项工作？

2. 在设定你所在团队的工作目标时，你起到了什么作用？你希望能够起到什么作用？你希望其他人能够起到什么作用？

3. 你的主管经理是否曾经有什么行为，使你怀疑他的诚实？如果有，描述一下当时的事件。你的怀疑最后有何结果？这对你的工作有何影响？

4. 你是否可以确定，在你所在的开发团队中，谁是任务权威，谁又是协调权威？你的团队中是否有一位“团队妈妈”？如果你就是这位“团队妈妈”，请介绍一下你担任这个角色的经验。

5. 用一张或者多张图表，画出你所在团队的正式的及非正式的组织结构。你是否能够将这些图表上的结构与你正在参与开发的程序的结构联系起来？你是否可以将它们与其他的因素联系起来？

6. 给出下列事件发生的原因：

a. 团队的一位领导者被更换；

b. 有一位新成员加入你所在的团队；

c. 一名成员离开你所在的团队；

d. 有危机开始影响到你所在团队的社会结构。

描述一下当变化或危机随时间发生并首次被公开之后，团队成员们的反应。描述一下作为变更或者危机的后果，团队行为的永久性变化——无论是更好还是更糟。

参考资料

如果读者有兴趣去进一步借助社会心理学进行研究，那么下列这些书籍将使你受益匪浅：

- Allport, F. H., *Social Psychology*, Boston, Houghton Mifflin, 1924.
- Asch, S. E., *Social Psychology*, Englewood Cliffs, N. J., Prentice-Hall, 1952.
- Lindzey, G., ed., *Handbook of Social Psychology*, Reading, Mass., Addison-Wesley, 1954.
- Krech, D., R. S. Crutchfield, and E. L. Ballachey, *Individual in Society*, New York, McGraw-Hill, 1962.
- Jones, E. E. and H. B. Gerard, *Foundations of Social Psychology*, New York, Wiley, 1967.
- Parkinson, C. Northcote, *Parkinson's Law*, Boston, Houghton Mifflin, 1957.

有些人担心，制定长期的计划只会延长项目完成的时间，这些人常常会引述Parkinson的话。那些借助于Parkinson的话来支持自己幼稚观点的人，应该花些时间通读一遍这本书，也许读完后他们将会获得一些新的认识。

□ Kohn, Hans, *Nationalism and Liberty: The Swiss Example*. London, George Allen and Unwin Ltd., 1956.

这是一本导论，它面向的读者是那些对民主的可能有所觉悟的人、那些曾经遗忘了这些可能的人或者那些从没了解这些可能的人。在读完这本书后，你将会开始对自己进行再教育。

□ Mills, H. D., *Chief Programmer Teams: Techniques and Procedures*, IBM Internal Report, January 1970.

但愿 Mills 将会在一般性的文献中发表他在“Chief Programmer Teams”（首席程序员团队）一书中的观点，从而使得这些观点可以与诸如“无私式程序开发”等观念做一比较，并且经受实验和观察的考验。该书主要依据的试验对象是外科医疗团队，用“首席程序员”替换了“主治外科大夫”，外加提供支持的一位后援程序员、一名程序资料库管理员以及可能的附属程序员，还有分析员、技术文献书写员、技术员以及其他专家。Mills 宣称，这样的一个组织机构“在程序开发项目过程中，会准许新的管理标准和新的技术标准得以应用”，但是鉴于外科学被认为是在一个正在倒退的技术领域中倒退得最严重的一个学科，所以人们还是怀疑将程序开发与外科医疗进行类比的合理性。另外，只要私下里认识几位外科医生，你就会怀疑这样的一个组织机构到底能对社会有多少贡献——对于那些总是自诩为“首席程序员”的人来说，也许这还不错，但是对于其他的工作人员呢？[①]

□ Mendelssohn, Kurt, A Scientist Looks at the Pyramids, *American Scientist*, V. 59, No. 2 (March-April 1971).

社会结构总是很明显地受到工程项目结构的影响，反之亦然。在这篇引人入胜的文章中，Mendelssohn 从社会结构的角度，研究了 5000 年前金字塔的建造如何导致了政府的出现，以及政府的出现又如何反过来导致金字塔的建造。这篇文章值得每位项目主管一读。

① 此处对“外科手术队伍”的架构提出了质疑，与稍后出版的《人月神话》（*The Mythical Mon-Month*，1975 年第 1 版，1995 年第 2 版，2002 年由清华大学出版社引进出版其中文版）一书中对“外科手术队伍”架构的推崇似乎有些矛盾。但《人月神话》一书中，作者面临的是 OS/360 这样的大型复杂操作系统及项目，在这样的项目中，将一个软件组的编码工作分成好几份给每个组员是困难的。因此，在这样的项目中采用“外科手术队伍”架构具有其合理性。——编注

第5章　程序开发团队

评　　注

“当然，一支程序开发团队之所以成立，是为了承担并完成某项由任何个人都无法独自完成的任务。”（p. 97）

虽然在我做出上述表述时，计算机程序开发尚处于其“黑暗时代”[①]；时至今日，虽然我依然同意这一提法，但是我对“任务需求”的理解更加宽泛了，还包括进来了团队及其成员的能力发展。经过这些年后我已经认识到，对于发展能力的需求同样无法由某个人来完成。通过与他人的交流和沟通，我们可以学习得更快、更好。

“虽然采用这些取巧的办法也许可以保证目前这一任务的按时完成，但是其前提是所有的子任务都按计划完成，然而这种情况非常罕见。”（p. 98）

时间是最具智慧的，它让我为自己的这句话感到惭愧。使用“罕见”一词过于谨慎；**总会**有些事是错的。由于像我这类作者的这等犹豫不决，使“乐观”估计的神话——“如果”一切方面都不出问题，那么一个项目需要多长时间完成——得以延续下去。如果要使这样的“估计”变得有意义，恐怕只有在项目进行的过程中勒令所有的热力学定律暂时失效。

① Dark Ages，黑暗时代。欧洲中世纪史，约公元476—1000年之间。——译者注

> “无论具体情况如何，程序开发团队的规模和组成似乎都符合这样一条基本的规律——如果希望通过最小的代价获得最佳的开发效果，你必须找到尽可能出色的程序员，并且给他们以尽可能长的时间，这样你需要的程序员数量也将最少。”（p. 99）

这条规律的真实性，已经被我后续的所有实验所证明。我还对它作了三个方面的进一步修正：

1. 所谓的“最小数量”，并非是指一名程序员；

2. “尽可能长的时间”中必须包含组建团队的时间。所以，如果能够找到现成的有历史记录的开发团队，那么你将会发现有诸多好处；

3. 如果你不是等到真正关键的问题出现，需要有人来加以解决的时候，才让开发团队开始工作，那么你又将发现还有更多的好处。

> “……当前最为常见的情况是：雇用一群实习生，并且给他们施加沉重的压力，同时又不提供必要的监督管理。在任何情况下，这种程序开发方式都必然是糟糕透顶的。”（p. 100）

时至今日，有经验的程序员到处都是，所以由清一色的实习生承担的项目已经很少见了。而现在又出现了实施程序开发项目的另一种糟糕的方式：雇佣一支由参差不齐的承包人（这些人实质上就相当于实习生）构成的杂牌军，然后用沉重的压力驱动他们进行工作，同时不提供必要的监督管理——或者反过来，过分地进行监督管理。

> “就程序开发而言，开发团队针对其任务的组织形式，最主要地决定于以下两个因素——目标系统的组织结构以及团队的人员构成。”（p. 100）

这种情况仍然与25年前一样，但是我觉得有一点与我原来的观点不同，我现在开始认识到：把软件产品与开发团队的人员构成进行匹配的做法，不再是不合逻辑的了。关于团队的人员构成，我的意思不是泛

指个人的技能，而是个人的人格个性。在今天的一个真正的开发团队里，因为随处都有现成的成果可以共享，所以个人的技能就显得不甚重要了。虽然如此，在这些年里我所见到的在团队组建过程中最常见的问题就是：在为团队添加新成员时，首要考察的标准是一些假定的个人技能，而不是该新成员与现有的其他成员共处合作的能力。个人的技能的确十分难得，但是和与他人的共处合作的能力相比，这些技能还是相对更容易（通过聘请或培训）获得的。

那个用来说明任务目标的设定与接受的（从 IBM 704 到 IBM 709 移植的）实例，从现在看来当然已经是很久远的历史了。但是在这些年之间，我又听说了许多发生在其他软件移植过程中的同类故事。在 1993 年，有人曾经问我，我所讲述的这个故事，是否实际上发生在一个把某个应用从 Microsoft PC－DOS 操作系统向 Microsoft windows 系统移植的项目中。这位提问者坚信，它曾经参与的一个项目就是我所提到的那个项目，只不过是出于保护当事人的考虑，我对那老式计算机的名字来了个偷梁换柱。

即使到了 1998 年，在一支开发团队中各成员就共同的目标达成共识，仍然与 1971 年时一样重要；这种情况也许将持续到 2021 年。但是这并不意味着，开发团队中的每一个人都必须与所有同伴们的目标完全相同；实际上，只需要找到每个成员不同工作的共同点即可。

在另一本拙著《成为技术领导者》（*Becoming a Technical Leader*）中，我对技术型的领导管理概念做了推广。在经过了 12 年之后，这本书仍然十分畅销，我想，这个现象应该可以最明确地证明：我关于领导能力之重要性的理论是千真万确的。当然，下述事实也许可以成为对此的一个更加明确的证明：在 20 多年来我们的 Problem-Solving Leadership Workshop 中，我的领导管理概念已经经受了来自世界各地的各个软件机构领导人的测试。

但是在关于开发团队领导管理的一节中，有一处影响极坏的、政治性的错误论断：

> “……在每支开发团队中安排至少一名女性成员。”（p. 119）

我必须解释，之所以在文中出现这样一句带有男性至上主义者口气的话，是因为在25年前，我曾在自己举办的高级软件开发培训班中，就女性程序员的问题做过一次调研。当我向学生们了解他们在开发团队中与女性的合作经验时，没有一个人能够回答——因为，**没有任何人曾经与任何一位女性程序员共事过**。在20年前我们的PSL培训班中，通常20名学员中仅仅只有4名是女性；而在今天，20名学员中大概有8～12名女性。

我的一名学生曾经对这一性别过渡阶段做过一次生动的观察："即使是在12年前，程序开发团队中绝大多数都是男性成员。我仍然记得在我曾经工作过的团队中，我所遇到的第一位女性开发人员。在那个团队中，只有2～3名男性同事会与她交谈。有的男性程序员甚至抱怨说，在团队中有一位女性在身边出现会分散他们的注意力。最后我才弄明白，对于这些人中的大部分而言，其实无论是在任何环境中，只要有女性在周围出现，他们的注意力总是会分散；而他们之所以不与这名女程序员交谈，是因为从初中起，女同学就没有与他们交谈过。"

当女性程序员很少时，很容易用老一套的方式来对待她们。但是，一旦在团队中出现女性不再是什么稀罕的事，她们就会发现自己其实能够胜任任何角色。有的时候，女性可能担任开发任务的主管；有的时候，可能是负责协调的主管；有的时候，又可能是这对某个专题的专家。与这种用人唯贤的方式相比，我现在仍然能够看到的不同，只体现在主管们官方指定团队领导人的时候。在这点上依然很不幸，男性仍然拥有至少二比一的偏见优势，这样的偏见使得许多团队不能按照可能的最佳结构来组建。

就应对开发团队的危机而言，我现在所了解的情况要比写作本书的时候更多。对我来说，与家庭临床医学家Virginia Satir共同的研究是关键的一步，我所学到的那些处理危机的方法，都被收集在《质量软件管理》系列丛书的第三卷中。尽管如此，最重要的教训就是：一般来说，预防危机要比处理危机容易得多——这也是我在第四卷中讨论的主题。

第6章 程序开发项目

与两个程序员必须共同工作时一样，当两支开发团队需要进行协作以最终完成某项开发任务时，协调机制的建立也就提到议事日程上了。单个开发团队的组建过程一样，这个阶段和同样会出现新的社会关系。比如，在这个阶段两位程序员之间的交往，既有作为两个个人的一面，也有作为各自所在团队成员的一面——尽管他们作为个人的工作目标可能非常接近，但是他们各自所在团队的工作目标却可能迥然不同。对团队中任何一位成员成功大小的衡量，不仅要相对于团队内部的其他同事，而且还要把他看成其所在团队的一部分，将其所在的团队与其他团队做一比较。现在，不同团队之间的协调需要一个二级的——或者说，更高的——领导管理；于是，就会出现这样的现象：与项目中最基础的工作直接相关的不一定是领导者的手下，而可能就是领导者自己。

在很多已出版的书籍、论文中，已经从管理的角度对开发项目管理的诸多方面进行了探讨，而我们并不想在此就整个问题进行综述。我们在下面几个小节中将要讨论的，只是从中挑选出来的几个方面——之所以这样选择是由于考虑到，对于程序开发项目管理思想的某些学派来说，它们可以作为一种适宜的心理治疗（或者也许更确切地说，校正）手段。

调整过程的稳定性

大型组织机构最令人感兴趣的一个地方就是，经过很长的时间它们依然可以生存下来，而这段时间要比任何成员呆在其中的时间长得多。在诸

如美利坚合众国这样的组织机构中，这个特点再明显不过了——毕竟，还没有谁能活到200岁。因此，任何人无论有多么伟大，也不能代表美利坚合众国。无论是华盛顿、林肯、杰斐逊，还是乔·麦卡锡，都只是美利坚合众国历史中的一小部分。但是，虽然这个道理再明显不过了，许多主管却不能在其负责的程序开发项目中对此加以运用。

即使是在开发团队中，我们也经常会看到这样的情况：虽然团队成立之初时的成员已经一个都不剩了，但是作为一个团队它依然存在着。即使有个别成员离开也仍然可以继续下去，这的确是团队这一组织形式的长处之一——因为毕竟，任何个人偶尔的离去，都不应该使正在进行中的工作因此搁浅。当然，团队的这种能力来自于其成员之间的交流，正是这种交流，使得在原有成员离去后，团队的整体目标和成就可以转移给接替的新成员并保留下来。从某种意义上讲，一个程序开发项目或者团队就犹如一条河流，虽然其中的水总是处于不断流动的状态，但是这条河始终还是一条河。

许多项目的主管都没有能够在自己负责的项目中领悟到这一点。在他们的理解中，整个项目的结构实际上更像是一栋房子——其中的任何一根木梁取出之后，整个结构都会坍塌。根据他们对待项目中某些人——尤其是那些所谓的“核心”人物——的态度和做法，就可以很明显地看出这种静态的认识；而这种态度的后果，往往是损失惨重的。

有两个例子可以说明持这种观点的后果。第一个例子的主人公是Mark S.，他又一次为一个实时工业过程控制系统设计了一个控制程序。鉴于他是惟一一个了解该系统工作过程的人，所以他被极力挽留下来以实现该控制系统。但是Mark认为设计该系统与实现它没有什么两样，所以他并没有为承担这项工作而沾沾自喜。他已经实现过两个类似的系统，所以对他来说，再实现第三个已经没有任何挑战性了。第三个系统与前两个之间有些许的差别，这曾勉强使Mark能有一丁点儿兴趣。但是，当他得知需要到很远的地方——一个工厂——去工作时，这一点点兴趣也丧失殆尽了。

Mark是个所谓的城市男孩，而他为了实现前两个过程控制系统，都曾经被要求在很边远的地方呆过很久。所以他的确对这种工作方式有点厌

倦了。在同一项目中的另一个部门里，有一位名叫 Bob A. 的程序员正在负责编写用于分析的程序。当时，Mark 所在的团队除了他自己以外，只有另外两名实习生。因为 Bob 极具开发经验，而且他也曾表示过对控制程序有兴趣，所以 Mark 请求把 Bob 调来，然后慢慢让他把自己的工作接过去，这样他就可以早日返回城市了。

然而项目主管却不这么认为。这位主管正在克服重重困难，以使更多有经验的人离开城市，到荒野之中来工作。所以，虽然 Mark 的建议显然有利于公司的整体利益，但是主管还是不愿意为了满足 Mark，而牺牲任何一名更有经验的程序员。他并没有把 Bob A. 调到开发控制程序的团队，而是给 Mark 再分配了另一名实习生了事。

可是 Mark 自己已经把团队中的所有工作都承担起来了，对他来说，原来的两名实习生已经是多余的负担了，所以他根本不想再要任何的实习生。他本来希望要么尽快完成工作，要么有人来接替自己，然而新来的实习生看起来只能延缓他的工作进展。他再次请求把 Bob A. 调来，Bob A. 同样也对不能获得这个机会感到不悦；但是主管却丝毫没有退让的意思。虽然他认识到，Mark 是自己项目中的一个“关键”人物，而且 Mark 正在闷闷不乐，但是他却并不愿意陷入到为 Bob A. 寻找接班人的麻烦之中。为了能够使 Mark 高兴起来，有一天他召见了 Mark，并且给了他一次可观的加薪。但是还没有等 Mark 按照新的标准领到头一个星期的工资，他就卷起铺盖返回了城里。

主管们经常会吃惊地发现，在获得一次可观的加薪之后，许多程序员反而会很快地——有时甚至是立即——拂袖而去。其实，在这种行为背后至少有两个心理学方面的充分理由——要是那些主管有朝一日明白了这一点，那么就应该对自己此前的吃惊感到吃惊了。有的时候，程序员会把加薪理解为他们将承担额外的职责，但是并非每个人都愿意承担额外的职责。程序员的推理逻辑是：如果我获得了更多的金钱，那么我也将承担更多的任务。一旦其承担的工作已经超出了自己愿意承担的极限，那么这时的加薪必然是毫无意义的，而且反而只会使他打退堂鼓。

然而更为常见的情况是，加薪会被理解为试图用金钱来替换程序员真正需要的某些东西。虽然程序员或许会盼望着能成为名义上的团队领导

者，但是突然的加薪会给他一种暗示：金钱上的奖励似乎是一种前兆，它预示着团队领导者的职位将被授予另一个人。在 Mark S. 的例子中，他对加薪的理解是正确的：加薪的目的是为了避免给他配备一名自己所中意的合格助手，同时也是为了将他与项目更紧密地捆绑在一起。他已经感觉到，他之所以呆在这里是为了支持主管，但是既然主管已经再一次地拒绝了他合情合理的请求——并不是为了立刻离开，而不过是为了获得培训一名接替者的机会——所以他只好放弃。

请注意在这个例子中，这位主管本来是想要把 Mark 与项目绑在一起，然而实际上却更快地把他赶了出去；而且，这样高的代价也没有必要。在 Mark 离队之后，实际上还是由 Bob A. 来接替他；但是由于 Bob 在此前没有接受过此项工作的培训，而且现在也没有人能够向他传授这方面的经验，所以整个项目只好搁浅了。最后，直到原定完成日期的一年多以后，一个东拼西凑式的系统才总算可以使用了。然而该系统的速度太慢，根本无法根据预知的变量进行控制。而究其原因，就在于进行控制的程序效率太低。结果，原来预期通过采用该系统达到的节约效果未能实现，在经过六个月的运行之后，整个系统就被扔到了一边。

还有另外一个类似的例子，虽然其结果不同，但是作为对“在持续的变化之中保持稳定性”的说明，还是很有启发的。在一个与空间研究有关的项目中，Henry F. 担任的角色与 Mark S. 类似。他设计了控制程序，并被任命为一支规模有点大、资历更佳的团队的领导人。当他对后续的实现工作感到厌倦后，他就把所有时间都用于培训手下水平最高的两名队员：Mary M. 和 Sid B.，以使他们能够像他那样地了解整个系统。得益于团队的民主式组织形式以及他们所推行的无私式程序开发方法，因此这些培训进展顺利。但是，这些培训却没有得到更高层领导的赞赏。

Henry 的一项工作，就是经常要向来访的达官贵人们做工作介绍，因为这是一个引人注目、众望所归的项目。Henry 的讲解每每效果很好，所以尽管他两度推荐 Mary M.，希望由 Mary M. 来接替他走上这个讲台，但是管理层两次都拒绝了他的请求。最后，他告诉 Mary 去准备一次讲解，然后在接待来访的那天佯装“生病”。虽然管理层的领导们有点手足无措，

但是也只好让 Mary 试试，结果 Mary 演讲的效果同样出奇的好。

经过这件事情之后，主管们才愿意把 Mary 接受为候补的领导者——实际上，他们已经开始让她做演讲——因此，Henry 觉得要求调换到其他工作岗位的时机已经成熟。但是，尽管 Henry 的主管已经明确地看到了 Mary 的能力，这位主管还是对 Henry 离开后的前景非常担心。他虽然承认 Mary 能力很强，但是更看重 Henry 是这一控制系统概念的最初创造者。此外，Mary 是一名女性；尽管主管知道她能够做出色的演讲，但他还是不相信，一位女性会有能力领导一个开发组。

对于主管的这种反应，Henry 或多或少已经有所准备，所以这时他又把 Sid B. 的名字提了出来。不幸的是，恰恰是因为 Mary M. 的成功，反而使主管更难相信，Sid 也有能力来带领开发组。Henry 提议给 Sid 一个机会，让他也进行一次演讲；但是主管却坚持认为：除了 Mary 之外其他人不能做演讲。于是 Henry 故伎重演，安排 Mary 佯装“生病”，从而使管理层终于明白过来。现在 Henry 可以离开了，因为 Mary 和 Sid 都已经可以——而且也乐意——接替他的工作。最后，虽然 Henry 获得了在项目中出任一个更高管理职位的机会，但这并不是他所追求的目标；他所希望的，是能够在程序开发方面从事对自己的想像力有挑战性的工作——所以，他还是走了。虽然此后管理层还是有点放心不下，但是整个项目还是顺利地进行了下去，直到最后取得成功。

这两件事之间的不同很值得深思。第一个例子中的失败和第二个例子中的成功，并不是由某个单独的因素所决定的。在与人相处方面，Henry 要比 Mark 更为擅长；他也更加幸运，因为他的团队中不完全是实习生，所以他不用经过上级管理层，就可以进行自然的工作接替；而且 Henry 的上级思想更为开化，也不刻板，并且或许其取得成功的压力要更小些；最后，对上级领导关于问题的看法，Henry 更有耐心、更能理解。Mark 的主管如果想要取得成功的话，不仅应该给 Mark 配备需要的助手，而且还应该支持 Mark 对助手进行必要的培训——因为 Mark 乐于担当一个不可或缺的角色。

从这两个以及上百个其他此类的事例中，我们可以得出的教训很明显。随着整个项目向前推进，人们会渐渐发现他们日益不满足于责任有限

的工作，或者——直言不讳地讲——因为他们认为其承担的工作层次低于自己的能力，并因此对工作感到厌倦。虽然并非每个程序员都追求更具挑战性的工作——如果被推进得太快，有些人就会离开——但是对大部分人而言，除非他们能够在工作中运用自己所掌握的知识，否则很快就会不满足于现有的工作。因此，为了在长时间内保持项目的稳定性，项目主管必须致力于按照一种“程序员处理工厂”的方式来运作项目——其一端是刚成长起来的实习生源源而进，另一端则不断地培养出经验丰富的领导者；而介于二者之间，所有人都随着流动的方向不断前进。

还有一个教训需要明确加以说明。任何项目都不会像由卡片搭建起来的房子那样，因为某个“关键”人物的离开而整体坍塌。至少，情况不应该如此；但是要是像经常发生的情况那样，如果管理人员确实这样认为，这种结果将被不幸言中。这正是从 Mark 的例子中可以汲取的教训之一，因为只要当初他的主管经理对 Mark 少一分依赖，就会事先扩大其团队的规模，而不仅仅是依靠实习生。无论如何，即使“关键”人物能够像一头牛那样地对工作永远满意，但是由于生病、应召入伍或者死亡等原因，主管手下的人有时也会不够用。如果项目主管希望稳定地推进自己的项目，他就必须严格遵守这样一条简单的格言：

> 如果某个程序员不可或缺，那么还是越快请他走人越好。

绩效评价

项目的规模扩大所造成的一个后果就是，对其绩效的评价变得更加困难。需要考察的方面更多，从而使任何个人都无法对所有方面同时兼顾，这是造成这些困难的部分原因。鉴于对程序开发进展的衡量是一个主观判断的问题，即使不考虑其他因素，针对同一个程序的开发进展，也会有不同的看法。然而，我们从没有机会去研究对同一个程序的不同看法。相反地，我们将通过对不同人对不同程序的看法进行比较和汇总，最终获得对整个项目进展的某种形式的概念。在做这种汇总的过程中，会引入大量可能导致心理方面的问题的因素。

在大项目中采用筛选的机制，将在其不同的阶段产生有趣的效果。这里有一个极端的例子：在一个军方的项目中，不仅需要进行程序开发，同时还要建造一个跨越世界范围的通讯网络。该程序开发项目的最底层由大约75人组成，他们被划分为12个开发团队；这12个团队的主管又组成3个开发组；这三个开发组的主管进而负责向一位程序开发项目经理报告。此外，在该程序开发公司内还有两个项目，分别是负责设计与建造中央计算机以及特殊硬件，所以这三名项目经理再进而组成一支团队，由一位公司项目经理率领。该公司项目经理同时还属于由来自不同公司的项目经理们组成的一支管理团队，由一名项目总经理来领导。

根据合同的要求，每个月都要向政府提供一份进度报告。鉴于这是一个资金浩大的项目，进度报告就很自然地要求用彩色打印。为此，报告的最终版本必须在截止日期——被报告月份的下一个月的10号——之前12天交到打印员的手中。比如，9月份的报告必须在9月28日前交到打印员手中；如果碰巧遇上周末，那么还要更早一些。

为了使项目总经理能够有时间来对报告进行审阅和修正，他必须在打印截止日期之前5天就拿到每个公司的报告。再考虑到邮寄的时间，这意味着各公司9月份的报告需要在9月20日前完成。在我们提到的这个公司，公司经理为了审阅手下3名项目经理的报告，又需要4个工作日。这样，程序开发报告就必须在15号之前完成。

这样倒推过去直到各开发团队，它们的报告竟然必须在**被报告月份之前一个月的倒数第四天之前完成**。也就是说，各个团队所报告的，实际上并不是当月的工作进展，而是对下一个月的展望。然而等到这些报告被汇总以后，就被打上了“进度”报告的标签——似乎没有人会关心其中的区别。

到现在为止，虽然还没有什么方面与心理问题特别相关，但是有两处微小的方面除外：虽然有些经过矫饰的事情是不可能的，但是人们却乐于相信它们；另外，在审阅者的级别与其消耗在审阅报告上的时间及其对报告实际做出的贡献之间，存在着有趣的联系——审阅者的级别越“高”，他要求“审阅”报告的时间就越长，但是实际上他对报告的贡献越小。并非什么工作也没有干——恰恰相反。无论报告的内容如何，它们每经过一

级合并，都被做了一些润色。

每一级别都是依照这样的方式对报告进行处理。如果一个下级开发组报告说取得了异常大的进展，审阅的人就会把指标稍微调低一点——因为万一下个月的运气不佳，这样留一手更能保险。反过来，要是报告说进展不大，审阅报告的人就会相应地调高一些——为的是掩盖团队的缺点，从而不致引起他人的注意。类似地，如果报告中列出的问题很多，他就会把不重要的问题删去一些；如果列出的问题寥寥无几，他就会对这些问题做详细的解剖，或者把一个问题分成两个单独的问题。

经过 6～7 级这样的筛选，最终定稿的每一份月度报告所显示的都是稳定而持续的进展：有若干方面稍微滞后或者超前；有若干从上个月遗留下来的问题；有若干新问题；还有若干尚待解决的问题。简而言之，在最高层完成的最终报告，根本没有反应出最下级单位的工作汇报。

当然，最下级单位“汇报”的，实际上不过是对进展的预测，因此在报告层层上交的过程中，做任何修改都是没有意义的。实际上，当开发项目中的一支开发团队的领导人，有一次碰巧看到了最后的报告后，他马上就领悟到了自己所提供的信息后来所遭遇的命运，于是他决定从此再也不浪费时间去把这些数字精确化。从那天起，他再也没有为索要精确的预测数字去打扰手下的程序员；相反地，他只需花上五分钟，就可以炮制出一系列令人赏心悦目的数字图表，然后交给上面。没过几个月，这种办法就在其他开发团队之间流传开来。虽然这样得到的最终进展报告与实际工作进展之间根本就不存在任何影响和联系，但是这种局面却一直持续了下去。

这个项目的进展报告，到底出了什么问题？我们可以确定出几个原因。首先，这里过分地强调对进展的**报告**，而不是对进展的**测评**。纵使逐层上报的速度加快到只需五天，仍然不会检查一下来自底层的原始信息是否具有意义。而且无论有些“管理信息系统”会做出如何诱人的承诺，只要其输入的是垃圾，输出的也**必定**是垃圾——尽管反过来不一定正确。此外，那些提供原始信息的人——各团队的领导者——迟早会意识到，没有人会对他们所提供的信息是否具有意义进行检查。实际情况是，只有在他们编造的数字过大或者过小时，其预测的数据才有可能被质疑。为了尽可

能地减少这种情况，各团队的领导者们已经学会了如何把上报中的极端数据消除掉。

有一个众所周知的心理学原理：如果要使学习的速度最快，必须向主体及时地反馈其表现之好或之坏到了什么地步。但是也许不为人所知的一点是：如果人们感到他们的绩效正在受人考评，同时又不能充分了解别人到底如何评价自己工作，那么他们就会通过不断变化的方法来试探这个评价系统。在像这样的报告系统之中，如果在提供了输入数据之后，处于最底层的人们没法获得相应的评价反馈，那么他们就会随心所欲地改变自己输入的数据，然后观察可能产生的不同效果——以期得到一些反馈，哪怕评价的结果很不理想。

在这个例子中，各团队领导者一开始就已经有点感觉出来：在他们提交的报告中，惟一重要的恐怕就是不要有任何令上级觉得异常之处。后来，等到其中一人碰巧看到了最终的报告，他们就完全可以肯定：上上下下所有人为之兴师动众的进度报告，不过是一个徒劳无益的游戏。而这样的一条偶然的消息，只是证实了众人的一个猜测——至少，它使大家都能节省出大量的时间。

从信息理论的角度来看，可以把这一逐层汇报的系统当作一系列的过滤器——每经过其中一次过滤，除了会有一定的延迟之外，同时也会有一定的信息损失。就这个意义而言，如果说信息是由一份报告所承载的，那么它主要承载的是惊人的消息。如果我们接到一份报告，称昨天有太阳升起，那么这份报告几乎没有包含什么信息；但是如果报告称昨天太阳没有升起，那么由于这是一个我们没有预计的事件，所以这份报告向我们传达的信息就要多很多。类似地，在一份进度报告中，最重要的信息会出现在那些有关巨大的或者微小的进展之处。但是这些方面的报告，在逐层上报的过程中会遭遇到什么样的命运呢？

在心理测试中，当要求被试在一根直尺上选择一点时，很少有人会选择尺子的两端。实际上，如果希望分出五个级别的差别，通常需要使用一支有7格或9格的尺子。位于尺子两端的格子——如果竟然有人选了的话——被偏向尺子的中心的极大值淹没了。也许这种效应可以解释我们在报告被逐层过滤过程中所看到的现象：每经过一级过滤，对应的审阅者就

会把经过此前的过滤还保留下来的极端数据进行处理，使之变得更平缓一些。

然而在这个例子中，每一级的极端数据都是相对于前一级编辑之后而言的。因此，随着报告层层上升，其中的数据也会越来越向中央集中。然而在这一处理过程中，每一级审阅者所过滤掉的信息，恰恰正是本来设立这一级审阅手续所希望得到的信息。难道没有办法解决过滤中的问题吗？

为了回答这个问题，我们必须首先确定一下，是什么心理上的原因，使人们倾向于把极值移往中心。一种可能的解释就是认知失调。如果某位主管相信，“出色的管理应该能够顺利、平滑地推进项目”，那么当他在实际中发现绩效出现波动时，就会认为这是与他上述估计的不一致。如果他的确这样认为，那么就很容易会接下来认为：这些波动只不过是手下人的错误判断，这些人“过于拘泥于其所处的局部环境，以至于不能对全局有正确的认识”。

另一种可能的解释——仍然不排除认知失调的想法——是由于暗示或猜测而引起的、来自上级的压力。根据管理手册中的正统观点，来自高层的压力既可以成为促进工作进行的推动力，也可能反而使整个报告系统失灵。主管们需要学会的，应该是如何促进人们修正自己的工作方式或者生产速度，而不应该反而让他们将自己的工作掩盖起来。根据这个原则，只有提供准确报告的人才应该获得奖励。但是在这里，报告的撰写人通常就是对报告中的工作负责的人，因此一旦某个人工作绩效很差，那么纵然他写的报告极为准确，也很难（即使不是不可能）因此而获得奖励。这种按层次组织的系统对主管们的目标要求是模糊的，正因为如此，我们才会看到一种“奇怪的”现象：随着时间的推移，在这种类型的组织机构中，报告系统的实际作用会越来越偏离建立这种系统时的初衷。

即使对不是逐层进行组织的项目，也同样可能有其他的原因，会迫使人们修改自己对项目进展的评判；这种压力尤其会来自同事之间。在Asch通过一系列经典实验揭示出：在来自周围人看法的压力影响下，人们将如何修改其关于两条线段相对长度的判断。在一个典型的实验中，有六名“被试”坐在同一张桌子后，然后要求他们做出判断，在一张卡片上的两条线段中，哪一条更长。其实，真正的、也是惟一的被试将被安排在

倒数第二个的位置，而其他五名“同谋”则已经在事先偷偷地被告知了问题的“答案”。在经过几轮正确的回答之后，所有列于那位真正被试之前的四个人都指鹿为马式地断言：那条更短的线段“更长”。他们的回答给被试造成了压力——虽然他确实能够看出前面的人都说错了，但是如果自己把事情的真相说出来，就必然会得罪他们。

已经发现，确实有些人不会屈从于这种“三人成虎”压力；但是遗憾的是，绝大多数人都会屈服。有一点大家都能够预测得出：两条线段长度的差别越小，就会有更多的人屈服——但是，即使两条线段的长度相差三倍，还是有一些人会屈从于周围人的意见。而且，正如我们根据认知失调理论所能预测到的：在事后接受询问时，他们都确实相信自己的判断是正确的。

这类关于社会性压力的实验，还有许多的变种。程序开发的主管们如果能对此加以研究，必将会大有收益。然而仅就我们目前的问题来说，能注意到这样一个事实就足够了：周围人一致宣扬的观点，会很容易地影响到我们对一些具体问题的看法；同时随着判断本身的难度加大，这种影响的效果就会变得更加明显。我们现在可以很容易地看出：在这样的环境中，像“程序开发项目的进展”这样困难而且主观的判断，会如何——在参加会议或者阅读周围同事报告的过程中——受到他人的影响。

这些实验还有另外一个相关的结论，这个结论告诉我们：当朝向错误选择的压力并非众口一词的时候，它又会对人们的判断产生什么样的影响。比如，如果要求第二名“被试”故意做出正确的选择，而所有其他的人依然选择错误的结论，那么这些错误的选择对那名真正被试的误导影响，竟然几乎会荡然无存。换而言之，哪怕只有一名“同盟者”，也会给人们带来足够的勇气，令他们能够认识到事物的真相。

在某个项目中发生过的一个例子虽然令人吃惊，但是却可以说明这种心理作用。在这个项目中，下一个星期就要对六个子系统进行集成系统测试，为此必须建立并派出一支由八个人组成的测试队伍。由于用于测试的计算机位于这个国家的另一端，因此维持这支测试队伍的开销极大。在测试开始之前，项目经理与六个团队的主管一起开了个会，在会议的结束之前，对各自的工作进展进行了汇总。每个人都确认：一切都已按照计划完

成，参与测试的人员将按计划派出，并且在他们驻扎的饭店里会合。在就各种技术的、后勤的问题进行过讨论之后，项目经理准备宣告会议结束。

“在我们休会之前，还有最后一项检查。”他说到，“关于下周的工作，是否有谁承担的部分需要推迟?”

虽然会场一片寂静，项目经理还是足足等待了 60 秒。最后，其中一位团队主管的手先动了一下，但是也许是因为不希望被别人察觉，他的动作很小。然而，项目经理还是敏锐地注意到了这个细节。“George，你有什么问题吗?”

George 的嘴唇嚅动了一下：“只有一个小问题。”

“有多么小?”

“需要稍微地推迟一点。”

“多久?”

“嗯。大约……六个星期吧。”

会场里顿时炸了锅。“六个星期?”众人异口同声地叫了起来。“我们都已经为系统的测试准备好了，而你的计划却要推迟六个星期。而且，你怎么能在整个会议期间干坐在那里，一言不发呢?”

但是项目经理却让其他的主管安静下来，然后向 George 表示祝贺，因为他有勇气在众人面前承认自己面临的困难，从而避免了在千山万水之遥的饭店里组建系统测试小组所需花费的巨额资金。经过进一步的讨论，项目经理说服了 George，他答应努力一下，争取在四个星期内完成自己的那部分工作。接下来，大家又一起重新制定了系统调试的计划。最后，项目经理再次准备休会。但是同样地，他又问了一次大家还有什么问题。

“嗯，”另一位主管极不情愿地插话道，“如果 George 需要额外的四个星期，我想……我的小组……也需要。”

“你的意思是，”项目经理问道，“你也没有准备好。”

“不完全是。”

“有多么不完全呢?”

“我估计大约要六个星期。但是我们会努力在四个星期之内完成。”

这样，洪水的闸门就被打开了。最终的结果是，六位主管中的每个人都承认自己的任务没有按计划完成。但是，要不是 George 率先承认了大

家心照不宣的这一事实，这次会议恐怕就不会发现任何问题的迹象了。果真如此的话，在这次会议之后，一次注定不会有任何成果的系统测试就会被启动，而最终只能白白地付出巨额的代价。

针对这种导致盲目的意见一致的社会性压力，哪怕是惟一的一个同盟者，也可以为释放这种压力提供一个安全阀。在认识到这个道理之后，在一些程序开发项目中，人们已经开始在他们的管理层会议中建立一种“魔鬼异议者”（Devil's Advocate）的制度。在通常情况下，由项目经理的一名技术助理职员担任这个角色。每次开会时，一旦出现众人意见“一致”的局面，就由此人来负责提出各种可能的负面观点。如果有人对公共的意见持有异议，但是却因为不愿触犯众人而默不作声，那么魔鬼异议者就可以为他提供一个思考的基准点。

如果魔鬼异议者的角色已经被制度化了，那么当这个人第一个站出来发表与众人对立的观点时，就不会像普通与会者那样受到巨大的压力。但是，其他人很难记得住他只是在扮演一个职业性的角色，而并非偏要事事与大家作对；因此随着时间的推移，职业性的魔鬼异议者也会渐渐感到周围的社会性压力不断增长。如果到了这一步，那么在他有效地扮演这个角色的能力应该提高时，这种能力反而会降低。为了避免魔鬼异议者角色被个人化，有些项目会采取在每次会议、报告之后，由各主管依次轮流担任这个角色的方法。通过这种方法，所有人都有机会体会到：从与众人相反的观点看待问题，并不见得会帮倒忙。

项目结构

在程序开发项目中，为了在进度评估时避开来自心理上的陷阱，一种重要的方法就是在工作与评估之间实行某种隔离。也许对于分层管理的组织结构来说，因为对任务开展实施控制的方向，与进度报告的流向正好相反，所以无法实现上述隔离，这是这种结构中存在的最严重缺陷。正如我们已经看到的，通过这样一个系统的信息流通质量，会逐渐地退化。为了克服这种退化，就需要对这种简单的层次结构进行调整。

在许多项目中，都设有一个标准小组，其最主要的职责并非是开发，

而是对其他小组的开发成果进行评估。在测试阶段开始前，该标准小组——或者其中一部分——就可能会承担起系统测试的职责；或者，可能从一开始就已经建立起了一个系统测试小组。除了进度评估以外，还针对其他的一些任务特意设立了专门的小组——资料小组、硬件小组、文档专家小组，还有系统程序开发小组（负责为项目开发所有要用到的服务程序）。在具有高度可见性的项目中，有一种类型的小组特别有用，这就是公共关系小组。通过这种小组的努力，可以把其他小组保护起来，以免受到来自外来好奇者的干扰。最后，还有很多为特定期间的特定项目而设立的形形色色的小组。

比如，如果在系统中存在一个程序错误，但是用常规的办法却无法找到它，那么就会成立一支特殊任务队伍，负责把这个错误根除掉。为了能够组建起这样一支临时性的队伍，项目经理必须在平时为项目保留一定数量的“闲散”人员，以便在出现漏洞时，能够利用这部分人力进行补救。无论计划做得多么仔细，任何项目都无法预测在实施过程中将遇到什么样的偶然事件，并且在事先就针对这些具体问题设立好对应的小组。因此，一个项目的结构在其生命期内经过多次重新组织，是情理之中的事。

在每次结构重组之后，不仅会“掀起一块大石头”，而且还会“发现许多虫子”。人们会拒绝从一个开发组调往另一个，也可能会拒绝与某个人共事，或者拒绝承担某项工作。而另一些人则可能坚持说，他们必须继续与某个人共事下去，或者必须继续负责某项任务，或者必须调到另外一个不同的开发组去。就其本质而言，所有的这些问题，其根源都在于人们为之效忠的工作目标各不相同——正如在一个人参与到某个团队中时引起的问题一样。

在一个项目中，任何人在加入到某个团队中之后，都会接受团队整体目标的一部分，并使之成为自己的目标。但是如果该团队是专门设立的，那么其任务目标就未必与整个项目的总体目标一致；或者，可能会包含其他团队的目标。在一些类型的社会关系中，这种冲突总是必然会出现。比如，与其他小组相比，有些小组的任务目标不那么受人尊敬、服务性更强些。例如，文档小组就经常为来自程序开发团队的成员所不齿——于是这些负责文档整理的人变得更加具有防御性，更加有门户之见，并且会尽可

能地避免与其他团队的成员打交道。而不幸的是，这种回避其他人的做法，正是我们不希望在文档团队成员身上看到的。

主管们却不会由于自己对这种对项目划分的鼓励态度而感到内疚。首先，他们就有自己的偏见，从而会允许这种划分在项目结构中得以体现。在一个大型的项目中，主管正在向一些来访者介绍他们的培训计划，以显示出他理解了在调整中保持稳定性的观念。他向来访者们出示了一张名单，上面是根据最终的评分排列的新的实习生；接下来，他提醒大家注意上面“所有”人都取得的好成绩。

“但是，”有一位来访者问道：“最后的三个人是怎么回事？他们的分数似乎要比其他人低得多。”

“噢，那是……”主管经理耸耸肩膀，解释道：“这没什么了不起。他们已经被分配去文档小组了，所以没有什么需要担心的。”

现在，如果其他的小组对文档员看得很低，不愿与他们合作，我们还会感到惊讶吗？

同样地，由于测试小组的任务就是批评别人的工作，所以这种小组与其他小组的关系通常也很紧张。我曾经有一名学生是一个程序测试小组的成员，针对其所在小组的成员与那些向他们提交程序进行测试的小组的成员之间的社会性交往，她曾经进行过研究。尽管他们的办公室是混在一起的，他们之间却几乎没有任何的社会性来往。每个小组的成员们不会与其他小组的成员共进午餐或者一起喝咖啡；甚至连借铅笔或者裁纸刀这样平常的事，在他们之间也不会发生。在同一个小组中成员内部的谈话中，其他小组的成员通常都会被描述成缺乏才智的、不友好的和不公正的。

在一定的程度上，这种相互讨厌的态度正是在管理层的纵容下形成的——因为每当这名学生想要为两个小组安排一次共同的聚会时，都会当场受到管理层的强烈抵制。这些人会说：这两个小组的人凑得太近，不利于整体组织结构的“健康性”。她解释说，在她所选的计算机程序开发心理学这门课中，她正在研究自己的一项假设；而之所以要举办这次聚会，正是为了测试一下她的这个假设。这样，她的想法才终于勉强获得了批准。举办这次聚会假托的名义，是为了庆祝该项目中的一个主要部分成功地通过了测试。在项目的前一星期，各小组之间的社会性交往增加了，这

次聚会本身也是一个巨大的成功。根据这名学生的观察，从此以后这两个小组在进行相互工作时，变得可以更多地通过个人之间的交流来进行，同时原来针对对方的冷嘲热讽也少多了。

至于在这种社会性环境改善之后，会使两个小组之间的工作变得更好或者更坏，我们没有办法来加以衡量。但是通过应用无私式程序开发方法的经验，我们确实知道，你的好朋友没有任何理由不能同时也是对你最为苛刻的批评家。关于这个问题，我们当然可以通过核对性实验来加以验证。与此同时，对于那些由于工作原因而自然产生矛盾的小组，项目经理不妨不要鼓励这种矛盾的发生。

在项目中自然出现的矛盾，并非都是沿水平方向发生的；有时，这种矛盾也会沿垂直方向发生——它把管理层领导者与实际工作的程序员们分隔开来。这种矛盾可能出现在设置目标阶段，因为项目的目标并不是每个团队各自目标的简单相加。项目目标通常都是自上向下设定的，早在大多数团队尚未组建成立之前，其中的主要决策就已经完成了。在这种结构中，团队内民主式的目标制定过程并不提倡。正因为如此，各团队的程序员很难会忠心于项目以及他们所在的开发组。

如果项目的管理层认识到，他们对项目的运作，是为了保证最终完成项目的一种组织机构形式，而不是为了满足个人的野心而不得不去攀登的金字塔；那么上述问题就可以得到解决。比如在一台发动机中，无论活塞还是曲轴，都不是气缸的“老板”。我们有那么多的项目在仿效层次化的组织结构，之所以这样，并非由于我们看出它是一种成功的组织机构形式或者自然结构，而是由于 19 世纪奥地利军队的成功。但是，恐怕我们很难想像出，有什么会比一群士兵与一群程序员之间的差别更大。

在老式的军队中，任何一名步兵都不是不可替换的。而分等级的组织结构之所以盛行，是由于被认为能够在各个可以互换的部分之间最快速而且最直接地进行协调。但是无论其外在形式如何，任何一项程序开发项目都不是一场战斗。因此，在项目中并不需要像在战场上一样强调通讯的速度；在战场上，为了能够在炮弹不断爆炸的背景声中仍然可以通过双向连接的电台互相大喊大叫，必须强调通讯的简单性；但是在程序开发项目中，这同样也是不需要的。因为在程序开发项目中，团队中成员的任务非

常专门化，相互之间的差异很大——因此他们之间所需要的，恰恰是缓慢而细致的沟通。此外，程序员并非是应召参加工作的；而且，当他的程序没有达到技术要求时，也不会被押上军事法庭。最后，项目中的官员既不必担心受到敌方炮火的伤害，也不必提防自己人从背后射出的黑枪，所以他们没有任何理由拒绝深入到战壕之中，与他的战士们并肩作战。

大型项目中共同的社会性问题

管理人员与底层的程序员在空间上相距太远，这是造成大型项目中出现许多社会性问题的根源。在一支开发团队中，不外乎有两种角色：程序员及其管理人员。但是在一个项目中，则可能出现第三种角色：管理人员的管理人员，或者称为第n级的管理人员。第一级管理人员与第n级管理人员之间的差别很大，因为第一级管理人员还有可能与实际进行中的工作有所接触，而对于第n层管理人员来说，他们只能通过其他经理，间接地了解实际的工作进展。

虽然比例很低，但还是有部分第n层管理人员可能最初也是程序员出身。即使对这部分管理人员来说，等到他们升至这样高的位置之后，原来所掌握的程序开发技能早已丢失殆尽。纵使他们为了保持其原来的技能，可能会不断进行更新，但是由于程序开发技术以及硬件的发展一日千里，今天流行的语言以及计算机很快就会无人问津，所以更高层的管理人员迟早会在技术方面落后。在一项对第一线程序员的小型调查中，只有15%的一线主管被认为与这些程序员的程序开发能力相当，而对更高层的主管而言，这个数字是零。据其中一些程序员反映，高层主管有时会到他们那里一边闲逛，一边看看他们正在做什么。但是对于这些程序员来说，这些主管想要借此学点什么的意图再明显不过了。当然，不会有人会冒险去得罪他们的上司的上司。然而，每当这些管理人员离开后，程序员们都会私下里一边拿主管们调侃，一边心中暗自发笑。

一旦管理监督人员不能吃透自己所负责的工作，那么他们在奖励程序员时所依据的标准，就不再是工作的效果本身，而是其工作的外在表现。如果有的程序员早晨能够很早就来上班，而有的则每每被发现在规定时间

之后才姗姗来迟，那么在主管们的心目中，对前一类程序员的评价往往会更高。有些程序员习惯于工作到很晚，但是由于主管们通常不大会注意到他们工作到很晚，所以这部分程序员往往不会成为被奖励的对象。同样地，主管们心目中关于程序开发工作的图景，是独自面对计算机屏幕上离奇古怪的符号、苦苦思索的程序员。因此，一旦他们看到有程序员与他人交谈，主管们就会认为他不务正业。

但是除此之外，主管还能做些什么呢？由于必须等上好几个月程序开发才会得出结果，他们也不能等到那时才去检查一下，哪些程序可以运行，哪些则不行——即使他有能力做出评判。或许，主管可以采用的最佳办法，就是依靠手下的一线主管们，并且强制要求他们去现场考察工作的进展——是的，甚至会要求他们去阅读程序。潜伏在程序员周围，然后突然袭击式地检查其工作——主管企图通过这种方式，给程序员造成一种主管时刻都在监视自己的印象。可惜的是，这一招并非总能奏效。

在采访中，有一位程序员告诉我们，他曾经通过炮制一个程序，并让项目经理自认为很了解这个程序的输出，最终“获得”了一次很大的提升。有一回项目经理突然造访他的办公室，这位程序员就捧出这个程序给领导“斧正”，“请教”在某种情况下的处理办法。项目经理确实提出了几点意见——即使那个程序是他们正在开发的系统的组成部分，这些意见也毫无价值可言，更不用说二者毫不相干了——然后就得意洋洋地离开了。几天之后在午餐时，这名程序员在餐厅中当着项目经理手下的各位主管，对项目经理的技术性建议“深表感谢”；果然，没过几天，他就被大大地提拔了一次。

主管们对工作提出直接指导是非常难得的事，因此，在属下面前因为自己的“技术性建议”而受到感谢，必定会大大提升项目经理的地位。在具体工作人员的层次，一个人的地位主要决定于其完成技术工作的能力——比如在调试程序方面传奇式的成就等。到了主管的层次，虽然他们总是不厌其烦地回忆自己当年的技艺如何如何精湛，但是一般而言，这种通过技术能力来提升地位的办法已经行不通。随着主管们在管理的仕途上步步高升，他们也会越来越不容易建立自己在技术方面的权威性。因此，对于那些能够给予他们某种权威的象征性地位，项目经理们会变得越来越依赖。

从身处系统之外的人们看来，这种象征性的地位非常有趣。在程序开发过程中的一些特定的地位象征，在其他的职业中不会存在；虽然这些象征在其他领域能够处理得当，但是在这里还是值得一提。比如，我们在办公室有卡式文件。在某些很低的阶层上，卡式文件就是一种高于一般程序员的正面象征——因为这意味着：这些文档异常重要，以至于其主人必须对其严加看管。当然，到了更高的阶层后，这些卡式文件就必须搬走，否则一旦被人看到，就会认为其主人与实际的程序开发工作靠得太近。如果在某个开发中心有一个机房只对少数人开放，那么那些被允许进入的人显然就要比其他人更为重要——实际上，让这些“重地”只对级别足够高的人士开放，根本不会出任何问题，因为这些领导在那里什么也做不了。在那些出入不成问题的机房，运行程序的优先级高低又会成为一种重要的标志——如果所有人都按照优先级别由高到低排了个队，在一个显眼的地方。项目经理看到，自己的工作将被安排在所有其他人之前，一种异乎寻常的满足感就会油然而生——尽管他可能从来也不提交任何计算任务。

后来，计算终端又成为了程序开发主管们用来象征其地位的首选标志。无论可怜的程序员们如何急需这些终端，每一位项目经理的办公桌上总是会摆上一台终端，而目的只是为了让每一位来访者看看。项目经理们到底会如何使用这些终端，这鲜为人知，或者至少可以说是一项严格保密的机密。根据终端类型的不同，偶尔来访的客人通常见到的是，一个东拼西凑而成的演示程序，或者一个名为 Nim 或者 Spacewar 的游戏。这类演示程序中最为漂亮的一个，会在屏幕上显示出一张由一些项目功能组成的 PERT 图[①]。项目经理只需轻轻一按，其中的关键路径就会显示出来，并且被染上比其他路径更为明亮的颜色。接下来，项目经理还可能会用光笔在其中某个节点上指一下，该节点就会被删除掉，同时整个图表会做相应更新。只要经理愿意，他可以再按下某个键，新的关键路径又将被突出显示出来。这个演示程序可以让来访者体会出经理对整个项目的生杀大权，而且屡试不爽。当然，这些都不过是骗人的把戏——所有这四个演示模式在每个系统中都是预先安装的，而且**只有**这四个模式。

① PERT：Program Evaluation and Review Technique。一种借助计算机技术的关键通路分析方法。——译者注

然而，有一种传统的地位象征方式却被程序开发主管们打入了冷宫，这就是秘书或者行政助理。这两个头衔的区别在于，秘书通常由女性来担任，而行政助理则可能是任何性别。有漂亮的秘书相伴左右，可能是一种地位的标志——尽管有些秘书确实其丑无比，但这同样也是一种象征，她表示出主人地位异常显赫，以至于只有这位**才貌出众的**小姐才能胜任其秘书的角色。

人们会把女性与办公室内的仆人角色联系起来，也会把漂亮的外表与智力平平联系起来。然而在大型的开发项目中，这些观念都将引起一种严重的社会性问题——性别问题。选修我的课程的一名女性学生，曾经在她所在的开发项目中，就男性程序员对女性程序员的态度做过研究。她的调查结果显示出，男性程序员会对职业女性产生不安全感，他们之所以对这些女性持鄙夷的态度，不过是为了掩盖自己内心的这种不安全感。随着年龄的增长，男性程序员中的这种倾向越发明显。而要是女性担任秘书就不会引起这类麻烦——其原因推测起来，不外乎是由于她们“知道自己的位置”。但是对于女性程序员来说，无论她们做什么，都会被有色镜片后的眼睛深深误解、曲解。

比如，如果看到一名女性正在与另一位女性交谈，人们会倾向于认为她们正在“背后议论别人长短”。如果这位女性正在与一位男性交谈，那么就是“卖弄风骚”。然而男性与男性却可以尽情地大谈特谈，不必担心会被冠以什么恶名——即使有，也只能是“业务”，无论这种“业务”到底是棒球、篮球，还是保龄球。在该项目的一个部门，只有两位女性。她们之所以被安置在同一支开发团队中，是因为经理认为她们会“喜欢与对方相伴”。经理每次见到该团队的主管，都会询问一下“她们”干得怎样。每个人都知道，“她们”指的是谁。

当然，多数的男性会认为这些问题不足挂齿。“男人很自然地会这样谈论女性，但是稍带友好地开个玩笑，还不至于对任何人产生伤害。”即使是那些与专业无关的女性，也会倾向于支持这种论调；她们同样会不遗余力地对其他妇女能够“承担这类复杂的工作”——意思是指“与男人们竞争”——表示怀疑。但是对于那些参与技术工作的女性来说，这类问题确是至关重要的——虽然她们不大愿意让那些好奇的男性同事知道这一

点。在许多项目中，女性被堂而皇之地排除在管理层之外；或者，至多担任第一级的管理职位。就这样，纵然某位女性可能对事业同样抱有雄心壮志，但是她最终不得不意识到，除非她离开这个项目，否则她未来的发展已经到了极限——不能不说，这对于激发整体的参与感是有害的。

毋庸置疑，男性经理们可以为这些歧视性政策搬出种种“合理的”借口，但是每一条理由都建立在性别歧视的基础上。他们可能会列举出个别的若干事例，来支持自己荒谬的理论，但是这些同样是出自于带有性别歧视的认识。为了说明苏格兰人的简朴，每个人都可以找到某个苏格兰人节省一针一线的例子；为了“证明”西西里岛人都是为非作歹之徒，也许你只需拿出一张印有此类消息的报纸即可；类似地，任何一位经理如果不愿意提拔某位妇女，他只需要举出一个例子，就可以“说明”在她被提拔之后，就会告假回家生孩子，或者跟随丈夫调动工作。然而，要是果真有哪位女性既不生孩子，也不随丈夫搬迁，那么经理们就会想当然地问道：“她还算是一个女人吗?”

任何歧视都会付出代价。在任何程序开发项目中，以“缺乏能力”或者其他的此类借口，免去任何人的职位，都会有损于项目的整体绩效。而且，一旦某类人开始意识到上级对自己的评判与其他人标准不同，他们的表现也就会不同。尽管开发小组互相之间的偏见同样也是一个严重的问题，但是针对妇女的偏见在程序开发过程中更为普遍，因此值得特别关注。程序开发人才及其管理人才的短缺问题，只需通过简单的一步就可以大为缓解，这一步就是：用完全同等的标准和眼光去看待女性——如果她们的确巾帼不让须眉。

小结

第二届北大西洋公约组织（NATO）软件工程会议在1969年底召开了，与前一年的会议相比，本次会议的口气已经大有变化。人，开始成为软件工程中最重要的因素。只要翻翻会议的论文集，你就不难找到答案。为了最明确的表述这个问题，这里我们将引述Joel Aron的评价：

我们对十来个项目进行了研究。虽然研究不甚正式，但是其结果却足以令我们相信：应该为程序开发的系统管理人员开设一门课程。

本项研究的目的，在于回答这样一个问题："我们的项目成功或者失败的原因何在?"我们判断"成功的"项目的标准是：能够按时达到预期的技术要求，没有超出预算，同时得到用户的认可。根据这样的标准，在我们所考察的10～12个项目中，只有一项是成功的，而其余的都以失败告终。

根据项目主管们以及曾在此前对项目作过评估的人们所提供的情况，我们对这些项目失败的原因进行了分析。他们给出了项目失败的各种原因。**但是就其本质而言，所有的原因都可以归结为管理上的失败。**我们之所以会遇到问题，并非由于我们缺乏这些技术本身，而是因为我们根本不知道如何对现状进行有效的管理。（文中强调为后加。）

好些年以来，我有幸能够与Joel Aron共事，其间正值他致力于向人们宣传他对程序开发项目的看法。然而不幸的是，计算行业中的大部分人仍然不能接受他的思想。就在本次会议论文集中紧接着的一页，才华横溢的C. A. R. Hoare说道：

> 从根本上讲，所有的问题都属于技术性的。如果你的目标明确，而且你拥有必要的技术背景，那么就不可能会在管理上出现什么问题。显而易见，在资源管理以及人事管理方面维持现状即可，仅此而已。

在某种意义上，Hoare所言不虚。然而遗憾的是，这些话并非总是正确。对于现在正在面临的问题，人们还没有搞清楚；而有些问题，人们甚至还没有意识到自己没有搞清楚——这本身就是管理方面的失误。我们曾试图证明，纵然拥有所有必需的技术能力，一个项目照样可能会每况愈下——因为，如果人们依然像Hoare一样不相信这些问题，那么在这个领域中的任何研究都注定会空手而返。就目前而言，支持这些问题重要性的只有若干琐碎的奇闻轶事；但是，我们必须找到其根本原因；也许，在此之前我们应该暂时先放下手头的各种程序开发项目。

思考题

请主管回答：

1. 你负责现在的开发团队有多长时间了？最初的成员中，有几个现在依然还在？请列出那些已离队人员的名单，并且给出其各自离队的原因。面对这种流动的格局，你是如何应对的？

2. 你是否遇到过在提拔或者升迁之后，很快就辞职的人？你是否认为，之所以对此人进行提拔或者升迁，是因为你没有或者无法给予此人另外一些东西，所以用这种提拔或者升迁来替代？此人辞职时提出的理由是什么？你认为真正的原因又是什么？

3. 你手下的男性或者女性中，是否有哪位对你来说是不可或缺的？如果有，那么假如她或者他不幸遇难，或者要是生病六个月，你会如何处置？

4. 在你所在的单位中，对于那些需要经过你、然后继续上报的信息，你是如何修改的？如果你把准确的信息上报给上级，你的上级会如何奖励或者处罚你？

5. 你因之而受到奖励最多的原因是什么？是因为提交了准确的信息？还是能博得上级欢心的那些信息？你向手下的程序员索要信息时，他们是否知道你的用途？这些程序员是否看到过汇总了他们所提供信息的最终报告？如果没有，为什么？

6. 在你的下一次管理工作会议上，如果出现了完全一致的意见，请尝试一下魔鬼异议者的方法。有一些人会试图通过非理性的论证，迫使你的思路纳入他们的轨道。记录下你当时所感受到的压力，以及这些人后来的反应。

7. 在你手下的开发团队中，你是否会因为认为某个团队更加重要，从而更加器重？你是否觉得这些团队要比其他团队威望更高？你手下的团队中，哪一个在士气方面存在的问题最大？

8. 为了促进开发效率的提高，你是否会鼓励不同团队之间由于兴趣矛盾而相互竞争？你如何测算出这种策略是否可行？

9. 环顾一下你的办公室，列举出所有可以找出的地位象征物。如果你找不到，请到其他主管的办公室看看，并列举出他的地位象征物。然后重新返回你的办公室，并重新尝试列出另一个清单。

10. 在你所在的单位中，女性职员受到的待遇是否与男性有所差别？在哪些方面有？原因何在？在你所在的单位中，女性职员的贡献是否不如男性？或者，你不了解这方面的情况？

请程序员回答：

1. 你是否有过这样的感觉：你很想离开某个职位，但是由于种种原因，却又不得不继续呆下去？你受到的这些压力来自哪些方面？对你在那个职位上的工作，这些压力有什么影响？

2. 你是否有过这样的经历：你本来希望得到某样东西（比如说一项不同的工作、更多的帮助、更方便地使用计算机，或者不同的工作时段），但是上司却没有满足你，反而通过给予你金钱来代替？这些金钱是否令你满意，以至于不再考虑原来希望得到的那些东西？

3. 你在自己的岗位上是否是不可或缺的？如果是这样，那么为了使自己显得更加不可或缺的，你是如何做的？为了使自己显得不那么不可或缺，你又会如何去做？

4. 你是否知道，自己所提交的进展报告或时间表后来的命运如何？你是否曾经怀疑过？你是否曾经询问过？

5. 你是否有过这样的经历：自己本来在技术方面有一项最佳判断，但是在来自周围同事或者管理人员的压力之下，你最后不得不妥协？请描述一下当时的情况，以及最后的结果。

6. 就工作本身的高低贵贱而言，下列工作在你心目中的先后次序是怎样的？为什么？

a. 文档化

b. 程序库维护

c. 系统测试

d. 进展报告

e. 标准化

f. 诊断程序开发

g. 培训

7. 你的办公室中有哪些地位象征物？参见请主管们回答的第 9 个问题。

8. 在你所在的单位中，对于女性成员的态度如何？如果你就是一位女性，周围人对你以及你的工作的态度如何？

参考资料

□ Naur, Peter, and Brian Randell, eds., Software Engineering, *NATO Science Committee Report*, January 1969.

Buxton, J. N., and Brian Randell, eds., Software Engineering Techniques, *NATO Science Committee Report*, April 1970.

在这两届国际会议中，云集了程序开发领域中的众多国际级领袖人物，他们就软件开发项目的控制问题展开了讨论。虽然这两份报告并不能覆盖会议所取得的全部成果精华，但是这几位编辑的工作还是可圈可点的。我们可以从他们的工作中得到关于软件工程的很多信息——当然，从行为科学的角度来看，其中的信息还很不够。

□ Asch, S. E., Studies of Independence and Submission to Group Pressure: I. A Minority of One Against a Unanimous Majority, *Psychological Monographs*, 7, Series No. 416, 1956.

这是一部关于不同观念与社会性压力的经典的和开创性的著作，每一位从事技术工作的人，都应该一读为快。

□ Kantowitz, Leo, *Women and the Law: The Unfinished Revolution*, Albuquerque, University of New Mexico Press, 1969.

这是一部根据时间的法律编纂史，对于那些自认为在美国，妇女已经受到了同等待遇的人而言，这本书是一剂很好的猛醒药。

□ Friedan, Betty, *The Feminine Mystique*, New York, Norton, 1963.

这是关于新“女性解放”的开山之作之一，这本书描绘出了在那些头脑清醒而且自觉意识得到提高之后的女性眼中，这个世界的样子。

□ Gagne, Robert M., ed., *Psychological Principles in System Development*, New York, Holt, Rinehart, and Winston, 1962.

关于20世纪50年代的“系统发展”运动，这本书是一份很好的工作文档。这场运动主要是军方的产物，它主要是通过RAND和SDC等机构推行，也包括一些大学的参与。除了从政府那里套来了巨额的资金之外，还不清楚这场“运动”是否曾经在其他某个方面取得过成功；而可以肯定的是，其对非军方工作的影响从此变得更加微弱。专辑中的工作层次紊乱，既有肤浅的陈词滥调、详尽的系统图表，也有一些照片，而且这些内容互有重复。其中“心理的”方面，主要是，把人看作是复杂系统中的部件——人力工程方法以及人事方法。对于计算机程序开发，其中没有任何有价值的东西；而本书的主要重要性在于，揭示了这种方法在程序开发过程中难以奏效的原因。

□ Meyer, Marshall W., Automation and Bureaucratic Structure, *American Journal of Sociology* 74, 3 (1968), pp. 256－264.

这篇论文是关于分别来自州级、国家级和城市级的254家财政机构的调查报告，它揭示出了各企业中数据处理部门与其他部门之间的结构性差异。尤其重要的是，一线管理人员要管理更多的人，更高层的管理人员则需要管理更少的人，这就造成了一种企业整体结构狭窄、底层宽广的结构。这种结构在底层把管理权分散化了，同时也拉大了最高层与底层之间的距离，这两个方面正是我们在计算机程序开发过程中忌讳的；在我们的环境中，第一线的主管经理对于手下程序员的具体工作知之甚少，同时沿着管理的层次逐渐向上，人员之间的沟通速度将会更慢而且变得不再可靠。

第6章　程序开发项目

评　　注

即使是经过了四分之一个世纪，新一代的主管们依然迷信：对开发项目的敬业精神可以用金钱来购买——每当他们不知如何来对开发人员论功行赏时，他们只会一味地涨工资或者发奖金。可悲的是，此类想法之所以久治不绝，是因为它确实行之有效。但是，开发项目的时间持续越长，金钱的效力也就越小。此外，一个人若能被金钱束缚住，那么他也可以被更多的金钱所解放。

有一些主管已经懂得了这个道理，为了把开发人员留住，他们现在正在考虑采用别的奖励方式。有的人甚至把开发项目按照小型规模来组织，各小型部分之间可以互相转换，从而不再需要——或者说不希望——把开发人员长时间滞留在同一位置。

这些年来，有无数的主管曾经对我表示感谢，感谢我在这一节中振聋发聩的警示。直到今天，这一警示依然正确：

> **“如果某个程序员不可或缺，那么还是越快请他走人越好。”**(p. 140)

这里关于绩效测量的讨论非常重要、非常适时，我已经把这部分内容加以扩充发挥，纳入了《质量软件管理》系列丛书的第二卷。心理学的测量方面没有什么变化。人们至今仍在玩着本章中提到的游戏——不断地虚构和谎报日程进度——因为和以前一样，在向上级汇报时，下级都会因为压力不得不投其所好，而不是汇报他们所负责项目的实际

进展。

在做了四分之一世纪的软件开发项目之后，我能够确定地讲：奥地利军队的模式依然十分盛行——至少在大多数主管们的头脑中是如此。虽然我相信在奥地利军队中，各级之间的负责分工很不一样，但是在今天，其形式更是大不相同。无论如何，有一个现象还是令人鼓舞的：现在已经可以自由地谈论或者评论其他的模式——尤其是那些强调团队协作的模式[①]，这些模式将团队协作看作一个项目取得成功的一个重要而核心的因素。

一个又一个的开发项目你方唱罢我登场，但是，其中的社会性问题依旧。我现在认识到，许多这类问题的根源都是一样的——缺乏自尊。治疗这种顽症的良方是：建立一种能够激发、而不是毁灭自尊的环境。

经过整整一代的更替，许多高层的主管在软件方面依然一窍不通。至少，我们现在已经拥有了一个繁荣的软件产业，借助这个产业，软件开发人员有机会来显示一下，自己可以在高级管理层上干得如何。其中不乏表现出众者，但是其他一些人的表现则说明，他们面临的管理上的问题并非来自于软件开发经验的缺乏。与此同时，这些年来的实际也明确地显示出，即使是毫无软件开发经验的人，同样可以成功地管理大型软件开发项目。所以，情况似乎表明，缺乏软件开发技巧根本不是问题。管理能力的缺乏，总是致命的问题；而软件开发经验的缺乏，不过是某些主管们很方便的一个借口——实际上，这些主管们真正缺乏的就是尊重。

随着硬件的发展，25 年之前显而易见的那些地位象征物，已是今非昔比；但是，用来显示地位的象征物依旧存在。这些变化表明，真正重要的并不在于具体的象征物是什么，而是人们对地位象征物的需求——一种由于缺乏自尊而产生的需求。

大体上，在开发项目中参与专业工作的女性已经不再被视为异类；

① 比如，可以参阅：Jim McCarthy, *Dynamics of Software Development* (Redmond, Wash.: Microsoft Press, 1995).

但是在部分人的眼中，她们仍然是同类中的下等人。尽管如此，我觉得这毕竟算是有了进步。但是也许你会认为，经过整整一代之后，也许我们完全可以做得情况更好。也许，正是因为有这么多男性并不认为这种进步还不够，所以我们应当做得更好。

在很久以前我曾经写道：

> “对于现在正在面临的问题，人们还没有搞清楚；而有些问题，人们甚至还没有意识到自己没有搞清楚——这本身就是管理方面的失误。……就目前而言，支持这些问题重要性的只有若干琐碎的奇闻轶事；但是，我们必须找到其根本原因；也许，在此之前我们应该暂时先放下手头的各种程序开发项目。”（p. 156）

幸运的是，我们并未死等研究方面的进展，我们实际完成的项目已经成为我们自己的研究工作的一部分。今天，在我们的知识库中，已经有了很多很多成功项目的例子，也有很多不成功的例子——有了这些数据，完全可以保证我们在软件开发方面的成功。

就我本人而言，最大的失误之处就在于：当时未能认识到项目管理中的情感因素。在这些年中，我已经在这方面作了一些补救，其成果主要见诸《质量软件管理》丛书；因此，现在我作为一名咨询专家的成功系数又提高了很多、很多。

Chinker

第Ⅲ篇

程序开发心理学

Computer Programming Psychology

第 7 章　程序开发任务的差异

第 8 章　人格因素

第 9 章　智力水平以及问题求解能力

第 10 章　积极性、培训以及经验



第 III 篇

作为个人行为的程序开发

在我的心中，能够与之相称的名称，只有自由——这种自由，存在于肉体的、智力的以及精神力量的充分发展之中，而这些力量潜藏于每个人的内心；除了由我们的天性所描述的自然规律之外，这种自由不承认任何其他的制约——确切地说，也就相当于根本没有制约……

——巴枯宁（Mikhail Bakunin）* ①

* Reprinted with the permission of Macmillan company。摘自：*The Political Philosophy of Bakunin*, by M. A. Bakunin. Copyright © 1953, 1964 by the Free Press.

① Mikhail Aleksandrovich Bakunin（1814/05/30－1876/07/01）。19 世纪俄国无政府主义者，民粹主义创始人和理论家之一。——译者注

在本书的开始，我们曾经试图对不同程序员工作绩效的差异做出解释。我们已经看到，这些差异在相当程度上取决于社会性的因素——这种程度，远不是我们随便考虑一下就可以了解的。然而，无论社会性因素对这种绩效差异有多大的影响，总还是存在一些其他的原因。任何两个程序开发小组，无论其结构多么类似，它们开发出的软件产品总是各不相同的。在这一部分的讨论中，我们将致力于从程序员个人的内因和外因来考察，以探寻导致绩效差异的其他原因。

我们如何进行这种探寻呢？关于从事程序开发的程序员，心理学的研究寥寥无几。尽管如此，我们还是可以假设已经做了心理学方面的研究，然后开始思考我们应该如何对其研究成果进行分析，从而就不同主体的绩效差异做出解释。

也许实验心理学家会首先问道："所有主体承担的任务都一样吗？"即使在一个精心控制的实验中，不同主体所进行的工作或者思考也难免会在不知不觉中夹杂进许多的差异。而在对两个不同的实验进行比较时，为了得出有意义的结论，我们就不得不需要首先考察一下二者在任务方面的差别。在对程序开发过程进行研究时，鉴于这里的每一项任务都与其他任务之间存在多方面的差异，所以我们研究的首要课题就是任务的差异。

心理学家接下来所关心的方面，取决于其实验的目的。如果是试图测量个体之间的差异，那么为了避开实验中具体条件的影响，他就要集中精力确保实验环境的所有方面尽可能地一致。比如，如果要对一些被试进行测试，他绝不会先在一个有空调的屋子里测试一部分人，然后又在另一个极热的屋子里测试另外一部分人——因为室内温度对人的工作绩效影响很大，相对于待测量的个体差异而言，如此之大的影响很容易把个体本身的差异掩盖起来。

但是，如果心理学家试图测量的恰恰正是温度对绩效的影响，那么他就必须人为地制造温度的差异。此时并不需要保持实验环境各方面条件的恒定，而是需要把个体之间的差异降到最低。为此，他可能会对同一组个体，反复进行多次测试——当然，此时我们同样需要考虑到实验个体在实验过程中的学习效应，并且采取办法加以克服。

无论如何，我们都需要同时关注环境的效应以及参与实验的个体之间

的差异，至于二者被考虑的先后次序，并不是重要的问题——只要我们做到恰如其分。

在接下来的几章中，针对心理学家们所谓的“个体偏差”，我们将其各方面因素加以分解剖析。换而言之，如果有两个人被指定在同一个环境中从事相同的工作，那么他们的行为差异将取决于上述因素的一种或者多种——当然，我们都知道，“完全相同的环境”是不可能实现的；但是，这种概念上的假设是必要的。我们所关注的个体偏差，可以进一步地按照一般性的方式被划分为“个性”、“智力”以及“培训”或者“经验”。

第Ⅲ篇　作为个人行为的程序开发

评　　注

对于原书中这一部分中的概念，我曾经寄予了最大的希望。或许正因为如此，我才不可避免地在这方面失望最大。程序员究竟是如何工作的？我撰写本书的初衷，就是为了激励人们对这一问题进行研究——就这个方面而言，没有什么值得我失望的。事实的确如此——在过去的25年当中，各种各样的有趣试验层出不穷，其中就包括在马里兰大学Ben Shneiderman实验室内进行的一些试验。

与众多的项目所犯的错误一样，我所犯的错误也是在一开始的时候设定了不适当的目标。我的目标其实并不在于激发出大量的试验，而应该是促使人们去理解这一问题，从而在个体程序员这一层次上获得更好的效果——然而从根大程度上说，这一目标并未实现。经过四分之一世纪之后，就其个体的技艺而言，今天的程序员与他们的前辈们没有多少差异。作为单独的个体，他们并没有变得更加适于借助人格进行程序开发，他们并没有变得更加聪明，积极性也没有什么提高，也没有得到更好的培训，工作经验也没有变得更加丰富。然而，他们的**开发效率**却的确变得更高了。

当我们还是年轻的“怪老头”时，我们究竟有多么聪明？当然，或许你会认为，正是因为我现在已经真正变成了一个古怪的老头，所以可能会对此持一种“扭曲的”看法。然而我要告诉你，对你的看法我（百分之百地）不敢苟同。我的理由有两条：首先，当年的我们并不比今天的程序员更加**聪明**。我成天都在与这些程序员们打交道，他们看起来与

我们年轻的时候真是太相像了——头脑敏捷、浑身充满激情。不时地，他们会表现得非常、非常地聪明；然而在其他一些时候，他们却又可能变得极为愚钝，令你简直无法相信。啊，是啊，回想起25年之前的我，与他们简直是一模一样。

其次，我也确实觉得，开发效率方面的差异，还可以通过其他的途径来加以解释——更便捷的工具、更融洽的团队工作气氛、更高效的管理体制，以及更好地理解程序员们将要完成的任务。



程序开发任务的差异

与“爱”一样，“程序开发”这个词所蕴含的行为方式也是无穷无尽的。中学生用 BASIC 语言在终端上的随意涂鸦，可以称作“程序开发”，但是工程师为专用的分时计算机编写尽可能紧凑的微指令程序，也被称为“程序开发”。除了名称相同之外，这两种行为之间还有什么共同之处吗？

有关程序开发的文献通常都假设：在中学生与工程师的行为之间存在某种共同之处。就其现状而言，这种假设也许是正确的。但是，我们对这种被称为“程序开发”的行为，没有做过细致的分类；也许，我们在探究程序开发绩效的差异时所遇到的困难，有相当部分正是由于缺乏这种细致的分类而造成的。确实，我们即使从 5 000 米的高度鸟瞰美国，也能够“创造”出一种关于经济活动的理论；但是，除了“人们在驾驶着汽车朝各个不同方向运动”之外，恐怕我们也得不出更多的结论。而为了解释行为中更细微的方面，我们就必须与地面贴得更近；为了排除程序开发中的种种神话，我们需要按照这种方法来做。

专业的与业余的程序开发

程序员这个名词的覆盖范围很广，而上述的中学生与工程师则是其中的两个极端代表。二者可能不同，也可能没有不同；一位业余程序员可能刚刚用六条语句写就了一个 BASIC 程序，可以用来求解二次方程的根，便开始就程序开发的理论与实践侃侃而谈——最令专业程序员们反感的莫

过于此。我们当然很清楚，在这种激烈对抗的背后，是因为这两种行为之间的差异没有为大家承认；而如果没有这种承认，专业程序员就会感到自己的地位被贬低了。确实，有些所谓的专业程序员，不过是用满口“之乎者也”的外表掩盖起来的雇佣文人；而有的业余程序员，通过编写一段小巧的程序，就可以获得关于程序开发的深刻见解。虽然这种情况的确存在，但是毋庸讳言，二者之间的差别依然是客观存在的。

也许，二者之间最为深刻的区别，就在于他们所编写的程序的最终用户。几乎无一例外地，就业余程序员编写的程序而言，其惟一可能的用户只有作者自己；然而，专业程序员总是在为别人开发程序。可以肯定的是，专业程序员也时常会为自己编写程序——生成测试数据、测试一个尚未尝试过的算法的性能、为仅仅两个实例命名等等。而且在实际上，每当他们进行这些工作时，专业程序员通常都会采用业余程序员采用的方法。但是，他们工作的主要方向，必然是由其他人（用户）对程序的要求决定的；这一简单的事实，从多个方面确定了其工作的性质。

正是由于业余程序员就是自己的程序的用户，所以到底是在程序开发之前还是之后进行构思，他们可以自由地选择。考虑这样一个例子：有一位学生需要编写一个程序，以求解二次方程的根。理想的方式是他坐在一个终端前——因为就程序开发后的思考而言，终端是再合适不过的位置了。他考虑到自己可能会需要一些输入。为此，他会为各个变量选取适当的名字。在这里，由于程序规模很小，而且它与其他程序完全无关，所以他自由发挥的空间很大——当然，他自己并未意识到这一点——他可以使用在自己头脑中最先出现的任何变量名，比如：a、b，或者 c。虽然这些很有可能会与其他需要遵守的符号或标准，甚至属性的声明相冲突，但是他只需依照在高中代数中所学到的概念，而根本不用顾及这些可能性。他确实不会顾及到这些，因为他所做的仅仅是编写出类似这样的东西：

```
GET LIST (A, B, C);
```

在通过类似的办法写完了程序的其他部分之后，他就可以开始尝试运行该程序了。在初始化处理阶段，终端会暂停下来，要求输入数据。由于是自己编写的程序，他心中很清楚每次应该输入的是什么数据，所以他用不着什么提示信息。这样，他就可以把下列进行预提示的语句省掉了：

PUT LIST（‘ENTER A，B，AND C’）；

同时，他也很清楚各个数据输入的先后次序；而利用该语言所支持的简化输入方式，他可以用下列不同的表格方式之一进行输入：

1 2 3

1，2，3

1，2，3

1.0，2.0，3.0

1E0，2.0E＋0，.3E＋1

此外，即使他不小心输入了：

1A，2，3

系统也会自动拒绝接受第一个数值，而他不必再编写一个错误处理子程序，也不必把数据被拒绝接受的原因打印到日志文档中。因为程序出自自己之手，所以他很清楚会有什么后果。

另外，在程序完成之后，他依然有机会思考；所以，即使是更为细微的问题，他自己也能够应对解决。也许直到输入了“1，2，3”之后，他才意识到这个输入可能会卡在后面的某条语句处——因为他接下来要计算一个负数（b^2-4ac）的平方根。但是此时自动诊断程序会帮助他——无须任何事先的深谋远虑或者编程——然后可能很快就能发现问题所在。也许他会意识到自己无论任何并不需要计算复数方根，因为这超出了问题的定义范围。只有在程序员可以自己定义问题时，这类的简化才是可能的；而如果是专业程序员遇到这类问题，他们就不得不暂且离开终端，然后与别人一起商定对问题的定义如何进行更改或者澄清。

一旦确定了需要计算复数方根，专业程序员就需要进行必要的整体修改；而对业余程序员来说，即使出现这种情况，他要做的事也要相对简单得多。在他完成了程序之后，他的任务也很简单。因为他完成程序之时，也往往是他忘掉该程序之刻。但是反过来，如果这种事情发生在专业程序员的头上，他就必须把修正部分打成一个包，然后发往这个冷酷的世界——这个世界给他的反馈，只有语气沉重而刻薄的意见，而为了进行后续修改，他们还不得不认真考虑这些意见。当然，作为专业程序员，他们

会在事先尽量把程序结构设计得合理，以避免大量的这类修正——但是在着手编写程序时，这是另一个他们必须考虑的问题。

作为专业程序员，即使其编写的程序不会受到他人的评价，他们也不会像业余程序员那样把这些方面的问题扔到脑后。比如，要是程序存储在终端系统上，那么在程序结束前，系统将会提醒用户释放所占用的存储空间。在这时，专业程序员必须保存其程序库中的每个程序，因为每个都可能再次被用到。但是业余程序员则不然，他们需要记住的可能只有一个程序，所以在这时，他们只需简单地命令系统进行删除——或者，甚至让系统自动来完成删除。但是，要是系统删除了专业程序员的程序库，那他们会感到多么可惜呀！他们绝不会这样。在系统的时限达到之前，他们会把长长的程序清单列出来，逐一进行核对；否则，他们的辛勤汗水就可能付之东流。

许多年之前，程序开发系统还只是初具规模，专业程序员与业余程序员之间的差别还不是那么明显。然而时至今日，业余程序员所希望完成的许多工作已经被隐含地集成在我们的系统之中，业余程序员与这些工作之间的距离很远，而且还在越来越远。但是荒谬的是，尽管二者间的鸿沟日益加宽，业余程序员们却越来越不能意识到这个鸿沟的存在，因为关于系统正在为他们做些什么，他们越来越不清楚。一位出色的主管手下的程序员可能根本不会意识到主管在管理方面所付出的劳动，这是这些主管所面临的一个问题；与此类似地，一位系统设计师的工作越是出色，其系统的用户越是不会意识到该系统的存在，因此这也是系统设计师们面临的一个问题。

说到主管，在程序开发方面，他们可能是业余者中的业余者。若干年前，曾经有一个公司决定给其行政管理人员开设一门课程，目的在于使他们能够意识到专业程序员所面临的问题。鉴于他们都是**行政管理人员**，所以在课程开设期间，公司为每个人都配备了一名专业程序员作为其“助手”。当每位行政人员都被要求“自个儿”编写一个程序时，整个课程达到了高潮。

为了加深行政管理人员对程序员所面临之问题的认识，在此任务期间还故意穿插进了频繁的电话打入、会议，以及对任务规范的小改动。最后

这些管理人员终于明白了——为了提高程序员的开发效率，他们应该尽力去把手下的人员“保护”起来，以免受各种干扰，更不应该自己就成为这种干扰的主要来源。他们同时也摒弃了另一个——更为根深蒂固的——观念。尽管周围有这么多的干扰，但是**经过努力**他们毕竟还是把程序实现了，难道不是吗？可以肯定，他们的“助手们”给了他们一点小小的帮助，但是这种帮助确实非常小。所以，如果他们能够按时完成一个程序，他们麾下的程序员们为什么不能？而且，既然他们可以只用一个星期就能够掌握，为什么会说程序开发那么难呢？

任何业余程序员都不会承认，在自己与专业的程序员之间有一道无法跨越的鸿沟；这些管理人员之所以会有上述的印象，与业余程序员产生错觉的原因是一样的。首先，由于来自语义上的错觉，把他们编写的“程序”与程序员编写的“程序”混为了一谈：前者不过是一个关于计算复合利息的问题，其实只要运用对数表或者计算尺，便可以更好地解决此类问题；而后者则是操作系统、编译器、工具软件，如此等等。其次，他们还有另一个错觉，这就是他们的助手“并未给予他们很多帮助”——他们之所以会产生这种错觉，是因为他们对程序开发的复杂性理解不够。他们的助手把他们屏蔽起来，正是为了使他们免于受到由这种复杂性引起的干扰；否则，他们就会由于无法完成他们的小程序，对程序开发产生一种不良的感觉。而且正因为如此，这种致力于使行政管理人员认识到程序员所面临问题的努力，恰恰得到了相反的结果。

然而，主管们很有必要更加充分地理解程序开发过程：可以用来说明这一点的事实是，主管们居然会天真到相信，专业人员经过数年的经验积累才掌握的东西，他们可以只用一个星期就能精通。专业与业余程序员之间的差别正在于专业程序员通过大量的经验积累之后获得的高超技能，然而每当你谈到这一点，人们总是会自然地当成是空洞的说教。但是也有人会主张说：差异的另一个重要方面，并不在于他们各自曾经编写过的那些程序，而是在于他们在**以后**将能编写出什么样的程序。业余程序员编写程序的目的，往往是为了得到满足自己要求的特定计算结果，所以他们只需把这一个任务完成即可。一旦遇到困难，他们想做的仅仅是如何加以解决——而解决问题的方式并不重要。但是，对于专业程序员来说，情况则

完全不同。他们首先考虑到的，是可以使用哪些现有的工具来绕开这个问题。他们甚至会采用其中的一种方法，以尽快地完成任务。但是，他们的工作并未就此停步。恰恰相反，工作才刚刚开始。他们首先必须搞清楚，为什么此前没能搞清楚这一问题——因为在未来的某一天当他编写另外一个程序时，很可能就首先需要对这个问题心中有数，所以必须在现在就做好准备。

也就是说，业余程序员所需要学习的，只是其待解决的**问题**；对于他们来说，所谓“程序开发方法的学习”，即使不是漂亮的幌子，也是一桩令他们苦恼的麻烦事。与此相反地，专业程序员确实要学习其**专业**——程序开发——而通过程序开发所解决的问题，不过是在他们水平提高过程中的一次偶然事件。

也可以从另一个角度来理解这一观察的结果：与业余程序员不同，专业程序员绝对不会很认真地来对待某个具体问题。以前他可能遇到过程序错误，但是将来必然还会遇到更多。这一态度上的差异，是引发两类程序员之间冲突的一个原因：业余程序员总是当着专业程序员的面，一边抖动手中的打印输出稿，一边振振有词地对计算机、操作员、系统、打孔编码员、程序语言，甚至政府大加指责——这些会令专业程序员感到非常厌倦与恼火。但是另一方面，业余程序员却对专业程序员的办事方式很不理解，因为即使眼看由于需要统计的平均值、方差等数据将无法按时获得，以至于其论文不能被收录到会议论文集中去，专业程序员也根本不会在意。

程序员想要做什么

对于对方所面临问题的复杂性，业余程序员与专业程序员没有充分的认识，正因为如此，他们之间会呈现出一种非对称的关系。虽然如此，专业程序员也会犯同样的错误，他们会嘲笑那些业余程序员，说他们的程序缺乏专业品位。业余程序员不能认识到自己与专业人员的差距固然是一个错误，但是与专业程序员所犯的上述错误相比起来，这种错误就显得多少可以原谅了。业余程序员的确可能所知甚少，但是一名专

业程序员如果是名副其实的话，也应该懂得更多。业余程序员不可能编写一个精致的错误处理子程序，因为他们不知道如何去编写，他们甚至可能不知道所谓的错误处理程序**为何物**。但是，既然他们根本就不需要这种功能，又有什么必要懂得如何编写呢？一名专业程序员，如果坚持把一个供个人使用的微小程序，当成需要被成千上万的人使用 5～10 年的操作系统一样来对待，这难道不是更加糟糕吗？

与任何其他的人造物一样，在设计程序时，也应该考虑到其特定的生命期以及应用领域。“助祭的杰作”（Deacon's Masterpiece）[①] 虽然“是按照那么合乎逻辑的方式制作的，而且也运行了整整一百年，然而毕竟也有……的一天”。与之相类似，在对程序进行设计时，其中的任何部分既不能设计过分，也不能设计不足。由于种种原因，程序中的有些部分对开发者的智力有很高的要求，而另一些部分则可能需要大量的精力；程序员们往往习惯于在前一类工作上投入过多的时间，这是程序员中的一种职业病。

下面是一个与此相关的例子：有一位物理学教授，要求一位半职业的

① 奥利佛·温德尔·霍姆斯（Oliver Wendell Holmes Sr.，1809－1894）最著名的诗之一。Holmes Sr. 毕业于哈佛大学，后任该校解剖学及生理学教授。其子小霍姆斯（Oliver Wendell Holmes Jr.，1841－1935）也毕业于哈佛大学，主修法律，后于 1902 年被罗斯福总统任命为美国最高法院法官，其思想对美国法律影响深远，故较其父更为知名。

该诗作于 1858 年，这里引述的是其中的第一节，原句为“... built in such a logical way. It ran a hundred years to a day,...”。此处的“the Deacon's Masterpiece”指的是英国乔治二世（George Ⅱ）年代由助祭制造的一辆单马双轮车。助祭在制造这辆马车时，挑选了最上好的木材，并且在设计时对每一零件可能出现的问题都考虑到了。1755 年 11 月 1 日，马车终于制造成功。不过，这辆马车的“生辰八字”不甚吉利——就在那天，葡萄牙的里斯本发生了大地震，共有 60 000 人丧生。尽管这辆马车一直运转良好，但是恰好就在其一百岁生日那天（1855 年 11 月 1 日）、恰好就在当年大地震发生的那一时刻（上午九点半），它终于在行驶途中崩溃，乘车的牧师“被摔在岩石上”，而马车则变成了“一堆垃圾”。

Holmes 在该诗插图版的序言中指出：“……大多数机械装置报废的原因都存在某种共性——比如据我看来，3/4 的马车之所以报废或出现事故，问题都是出在车轴与车毂的结合部位。因此在制造马车时，工人们应该首先确保这一部位的牢固，然后再考虑第二薄弱的环节，……如此下去。只有按照这样次序，才有可能达到助祭所达到的‘完美’境界……”——尽管他的“杰作”也同样没有能够经受住时间的考验。

Weinberg 在此处用助祭的马车暗比现代的计算机程序，意思是说：在设计程序时，程序某一局部的精巧程度并不是最重要的；就整体而言，只有对每一部分都进行恰如其分的设计，整个程序才不至于因为存在某个薄弱环节而过早出现问题甚至报废。——译者注

程序员编写一个程序，用来对矩阵求逆。由于在存储器中同时存放的矩阵有很多，这位程序员希望能编写一个子程序，每次从磁带上读取一个矩阵以供处理。但是，鉴于自己在输入一输出编程方面经验有限，所以他认定这是提高这方面能力的一个绝佳机会，于是他开始四处寻求咨询。

“为了对待处理的输入数据进行缓冲，我应该如何对来自磁带的输入数据进行编程呢?”他向一位更加专业的同事问道。正因为更加专业一些，所以这位同事并没有立即正面回答他的问题，而是反过来问他一个问题：

“你**为什么**需要对输入做缓冲呢?”

“当然是为了节省时间了。”

“你是否估算过，可以节省**多少**时间呢?”

“说不好，但是应该有很多，因为有大量的矩阵。”

“**多少**?”

“我说不准。很多。”

“大致估计一下，有多少?”

“**大概**有 100 左右。”

“好的。那么，每个矩阵又有**多大**呢?”

“10 乘以 10。”

这位同事马上在黑板上做了个运算，其结果显示出，读取所有矩阵只需要一分钟时间。

“你看，”半专业的程序员带着胜利者的口气说道：“是要花**很多**的时间吧。”

“也许是吧——或许也不是。你的程序需要运行**多少次**呢?”

“你是什么意思?”

“我的意思是，如果你准备编写一个缓冲子程序，你就必须对它进行测试；而我怀疑，你是否可以在一分钟的机时内完成测试。所以，倘若你只要处理一组矩阵，我奉劝你还是别去理睬这种事。光是计算机用来进行测试的时间，就要超出你所可能节省的时间——这还不包括你自己花费的时间。”

“但是有一点你不知道，”面对这样一个能够编写新潮而有意思的程序的良机，半专业的程序员并不希望就此错过，于是他说：“这必须是一个

高效的程序。”

他的同事虽然对他的反应感到失望，但是仍然反复地向他介绍自己的理由。但是，唉，他的所有这些努力都是枉然；经过一个学期，当他再次遇到自己的朋友时，这位朋友仍然没有能够让他的程序工作起来。而那位可怜的物理学教授，根本就不知道事情的经过，还在苦苦地等待他的矩阵数据——不过他也有值得高兴之处，因为有人因此对他大肆吹捧，极言其提出的程序开发问题是何等何等之艰深复杂。

这样的例子还可以举出上百个，其中所蕴含的道理是一样的：每个程序都有其特定的细致和复杂级别，这种级别取决于它的具体用途。从某种意义上讲，如果付出的努力超出这种级别，要比低于这种级别更为不专业。如果希望确定单独一名程序员的工作是否出色，我们就必须看看，他是不是按照恰当的级别来处理这个问题。有些人可以凭借其才气与个性，成为一名出色的业余程序员；但是从专业程序员的标准来看，这种才气与个性往往正是极不合适的。重要的是根据手头待解决的问题来调整自己的工作方式的能力，而一旦缺乏这种能力，任何人都绝对不适合做一名专业程序员。

尽管如此，程序员通常都不能真正了解自己需要解决的问题是什么，因此也就谈不上根据实际问题来调整自己的工作。换而言之，他根据自己的主观愿望做了一些**假设**——也许限于自己已经掌握的方法，也许是根据其完成的上一项工作而做出的臆测——但是，他总是要等到工作完成之后，才会认识到这次任务的要求到底是什么。为实现缓冲而进行的工作本身并非坏事，但是这仅仅是就某个特定程序的用途而言。如果矩阵变成100乘以100，而且其数量达到几千，那么我们故事中的替罪羊就可能会成为英雄。

关于程序开发绩效的研究屈指可数，但是这方面的研究都存在一个共同的问题：对程序开发目标中的奇异性缺乏认识。即使是精心挑选出一批程序员，而且分配给他们的任务也一样，在这样的条件下对他们的“绩效”进行测量仍然是不够的。心理学家都知道这样一个事实：在向每一位被试布置任务时，必须明确地交待具体的要求；而且，在大多数情况下，即使这样做也仍然不够。比如，给定一组单词，然后要求我们找出其各自

的同义词，那么我们将会把速度还是准确度作为评价工作好坏的标准呢？如果心理学家要求我们尽可能快速地完成任务，那么我们是否会听从他们的命令，而不管有多少因为粗心大意而造成的错误呢？这取决于许多因素——因为和一些人相比，总有另一些人确实生性谨慎。为了解决这个问题，心理学家们不知度过了多少个不眠之夜；或许，程序员及其主管也多少应该关心这个问题。

为了测试事先设定的目标对程序开发绩效的影响，我们曾经进行过一次小型的实验。在实验中，我们请4名程序员来解决同一个问题——这个问题不那么简单，如果他们每天投入1/5的时间，那么需要10个星期才能完成。这4名程序员分成2组，每组2人：问题定义的大部分内容都是用施乐打印机打印出来的，两组完全一致；但是在最后一页上，两个小组各自得到的内容不一样。前一组得到那一页内容如下：

> 你们在这个项目中的任务是：经过彻底的调试，实现一个尽可能高效的程序；所谓高效，指的是要求占用CPU的时间最少。如果有必要，你们可以在128K范围内，使用任意大小的核心内存。你们不必追求完成项目的速度，只要在第9个周末能够进行最终测试即可。

后一组接收到的最后一页内容如下：

> 你们在这个项目中的任务是：经过彻底的调试，在最短的时间内实现一个程序；只要不至于影响到项目的工期，程序的速度与存储空间效率等因素可以不必考虑。不过，你们在本项目中投入的时间不能超过正常的负担（1/5的工作量）。而且，你们必须把程序本身的大小控制在128K以内。

通过如此明确的指示，我们希望能够通过测量看看，对同一项目在目标上的不同理解，到底会导致哪些差异。每一组内的程序员并不知道，其他小组的工作目标与他们有所不同。但是等到项目结束时，不同小组的工作效果差别之大令人瞠目。被要求在最短时间内完成任务的那个小组，平均花费的机时只有另一个小组的2/5，而花费的人力时间只有另一个小组的1/3；但是另一方面，这个小组所实现的程序的速度，平均起来要比另一个小组慢10倍！（参见图7-1中的“索引问题”。）

我们使用另一个不同类型的程序，对另外两组程序员，也再次进行过这个实验；当然，这个程序的复杂程度差不多。在这里，实现特别大的节省的可能性要小一些；但是相同的是，被要求“尽快完成任务”的那个小组所花费的平均机时，仍然是对照组的 2/5，而人力时间仍然是对照组的 1/3。在这次实验中，要求“程序效率高”的那个小组所实现的程序，在速度上要比对照组快出 50%（参见图7－1中的“文件问题”）。

	索引问题		文件问题	
	平均运行次数	平均执行时间	平均运行次数	平均执行时间
高效的程序	78	1	60	1
快速的编程	30.5	10	27	2

图 7－1　环境对绩效的影响

这两次实验给出了很好的对比结果，这一对比显示出：如果要努力提高效率，那么效率可能提高的程度将取决于问题的类型等诸多因素。因此我们不可能笼统地讲，为了提高效率，我们应该尽多大的努力才是合理的；即使我们对所有问题的关注程度完全一样，也不能做出此类论断。但是通过这些实验，我们还是获得了很多有益的结果，其中重要的一点就是：程序员工作绩效的差异，在很大程度上取决于其对所有完成任务目标的理解不同。

对于强调“程序效率”的那个小组，如果逐步增加允许使用的机时以及投入的人力时间，以进行更细致的统计，我们将会发现：其效率在相当程度上将取决于在遇到始料未及的困难时，程序员采取何种应对方法。在强调“尽快完成任务”的小组内，如果某种方法不能奏效，程序员就会立即放弃它，然后转向另一种方法。然而要是强调“程序效率”小组的程序员遇到困难，他们会极不情愿去更换方法，因为这样很可能会牺牲程序的最终效率。综上所述，尽管在一开始时两个小组的程序员可能构思是一样的，但是强调“尽快完成任务”的程序员最终很可能使用一种完全不同的方法——不过，当他已经大功告成的时候，对照组中的程序员可能仍然在苦苦奋斗。

另外一点需要注意的是：困难的来源或多或少地与最终结果没有关

系。比如在某个时候，可能有一个由编译器错误而引发的某种困难，而这个困难又是在按照某种方案解决该问题过程中的拦路虎。尽管两个小组同样都会遇到的这个困难，而且（自然地）二者都对这个困难束手无策，但是在最终发现这是一个由编译器导致的问题之前，强调“尽快完成任务”的小组可能很早就已经放弃了这一方案，而另一个小组则可能会执著于这个问题很久，直到最后发现问题的症结所在时，才不得不咽下苦果。虽然这类事件的结果会使该小组觉得“天理不公”，但是在全世界众多的计算中心中，每天都会有成千上万的这种事情发生。从心理学的角度来看，教训是显然的。虽然任务目标的说明未必明确，但是程序员对任务目标的主观理解的确差异很大；正是由于这种差异的存在，同一个客观的因素（比如编译器中的错误、难度大的算法，诸如此类）对项目的最终影响可能迥然不同。因此，如果需要衡量程序员的绩效、语言的性能、操作系统的性能，或者其他方面，我们首先需要确定的就是：所有人正在为之努力的，是同一个问题。

当然，在实际生活中，通常不会有多个小组同时开发同一个程序。因此，我们通常可能不会意识到：开发组的实际工作目标，可能与其主管等人的理解不同。所以，我们可能事先没有确保：人们已经针对**所有的**任务目标进行了足够的沟通，并且这种沟通仍在不断进行。要是果真如此，那么等到最后出现程序未能满足日程安排、运行效率不够、耗费过多存储或者诸如此类的问题时，我们就不应该大惊小怪了。

还是让我们稍微停留一下，因为“实际情况”绝对不会如此简单。在人们就任务目标进行沟通时，甚至会存在某种危险：目标居然可能改变估计！在我们所进行的实验中，在了解了事先的明确指示对最终绩效有何影响之后，我们又转过头来，对程序员事先的估计进行检查。所有的程序员都对批处理任务的道数、完成每个项目花费的天数等做过事先估计。图 7-2对他们实际的与估计的任务道数、项目持续天数等进行了比较。很容易看出在此期间发生了什么变化。在被要求尽可能快地完成任务的那个小组中，成员们在估计完工时间时会趋于过分保守；而实际上，他们完成任务不仅要早于自己预期的时限，而且要早于另外那个小组——虽然后者在自己做估计时显得更为乐观。

	程序运行次数		天　数		
	估计的	实际的	估计的	实际的	延误百分率
高效的程序	22	69	48	76	75
快速的编程	39	29	68	65	25

图 7-2　环境对估计的影响

需要间接说明的是：在被要求尽可能快地完成任务的那个小组中，有一位程序员坚决拒绝对完工时限做出任何的预计。在项目进行的中途，这名程序员患了阑尾炎（虽然很幸运，他本来就是小组中可有可无的人）。在下一次确定项目目标时，程序开发主管们需要首先对这张图表仔细琢磨一番。如果目标是明确定义的，那么就会有两种效果：程序员们为了实现这个目标，可能会牺牲另一个目标；而且在事先对目标实现的质量进行估计时，程序员绝对不会保守（或者精确）。对没有强调的目标进行估计，恐怕会根本不可靠，这既有程序员对此不认真的原因，同时也是因为这些目标的不重要性，所以难免会为了实现其他（更重要的）目标而被牺牲掉。不幸的是，对于事先估计的这种修改，实验者没有任何思想准备，所以我们遗漏了让程序员对其程序进行估计的步骤。尽管如此，还是会得到同样的结果——只不过两个小组的结果会相互颠倒。

要是这个结果能够被证明具有一般性，就可以澄清人们关于帕金森定律（Parkinson's s law）[①] 的一些困惑，许多主管的噩梦也将从此结束。当帕金森说“工期确定后，工作的实际进度会尽量充满它”时，他所要提醒我们的正是：工作日程的存在会反过来影响工作的进展速度。但是在这里我们已经看到，正因为有了一种**以任务目标**形式存在的日程计划，这一目标会反过来对“实际分配的时限”产生影响。之所以工作的实际进展会相应地拖延，以耗尽整个分配的时限，正是因为有其他目标的存在，而这些目标相对于日程计划的重要性并未在事先被明确说明。或许，只要我们沿着这个思路继续前行，就能够搞清楚，到底是在什么机制的作用下，才会导致这样一个为人们普遍接受的结论：没有能够按时完成的程序开发项目。

① 该定律称：时限充裕则工作进度随之而慢，收入大则开支亦随之增多。——译者注

程序开发工作的不同阶段

我们必须消除的另一个错误理解是："程序开发"是一种整齐划一的工作，所以需要一批整齐划一的人才。至少对于专业程序员来说，要把定义说明转换成最终的程序，需要经过各种不同的工作；而为了完成这些不同的工作，就必须要有各种类型的人才。

在一个组织有条不紊的项目中，因为可以根据不同人的各自所长，分配对应的任务，所以任何一名程序员都不必（也不可能）具备完成整个程序的所有能力。实际可能发生的情况是：虽然在程序开发的某个阶段，某位程序员可能具备非常对口的能力，但是一旦进入另一个阶段，这些能力却或多或少地可能成为累赘。在对整个系统进行设计规划时，要求设计者能够把握全局；然而在进行调试阶段之后，则必须尽可能地了解到每个细节，以至细节的细节。在进行编码工作时，则经常需要把代码的冗余降到最低的程度；然而在进行文档化工作时，则需要把一个本来很简单的句子扩充为一段。

如果把成功的程序员放到管理的位置，那么他很可能一事无成。与此相类似，等到进入系统调试阶段，一位原本非常成功的设计师可能会变得碍手碍脚。反过来，虽然某个人在调试时可能会大显身手，但是在项目尚处于设计规划的阶段时，却由于他这方面的才能没有用武之地，所以可能要暂时让他先靠边站站。为了保证在需要的时候总能找到最合适的人选，我们必须首先将程序员需要完成的工作进行合理、细致的分类，而不是笼统地称之为"程序开发"。

程序开发，也常常被人们描述为这样一个过程，它从问题定义开始，然后进行流程图分析，进而编码，测试，直到最后的文档整理。这种粗略的划分虽然不无道理，但在很多方面却不尽合理。首先，实际工作的流程未必都是这样的一成不变，比如有时在测试、编码、确定流程图之前就可以进行文档整理，有时甚至可以提前到系统分析之前进行。其次，正如我们在把程序移植到另一台计算机上或者另一种语言中时的情况一样，并非每个步骤都是需要的。再次，这些工作根本不可能严格按照某个顺序来实

施——而且在实际工作中，确实极少会这样进行。有谁未曾在系统分析、流程图设计、编码、测试以及文档整理时发现问题后，反过来对问题定义作必要的修改呢？或者反过来，有谁曾经见过哪张流程图，在经过编码后不曾做过任何修改呢？又有谁见过哪段代码，在经过测试之后不曾做任何修改呢？

为了从心理学的角度来对程序开发做一研究，我们必须把这些复杂的行为划分成更为简单的部分——当然，这些部分都是我们已经建立过的。但是，考虑到程序开发过程本身的循环、迭代属性，即使是我们上述的分割也仍然是过于精细的。根据这种划分，各部分之间的界限非常模糊，甚至根本没有界限。如果我们向一位程序员询问他正在从事的工作，那么可以肯定的是，他会毫不犹豫地回答说在“编码”或者“调试”。而且，所谓“进展报告”的制度虽然在多数组织行之有效，但是这种制度会驱使人们不顾实际的情况，把自己的工作归入生硬的条条框框。人们就这样被逐步地诱导，以至于最终相信：他们的工作确实可以归于自己每星期所填写的进度报表中的某个类别。

作为一个特殊的实例，请设想一个这样的项目：其中为所有任务都设定了一个必须完成的时限。比如，在临近问题定义完成的时限时，显然大家都会疲于拟定最终的“任务定义书”。而到了时限的最后一天，所有的定义都将送到项目经理的手中——虽然到了这个时候，也并不意味着项目的规范已经被定义好了。实际上，这仅仅意味着：所有定义都汇总到了项目经理的手中。

现在，任务的定义已经在形式上完成了，接下来将开始进行系统分析。也许正是由于任务定义是在仓促间完成的，所以定义中的一些纰漏很可能会在这一阶段暴露出来。要是果真如此，接下来就会有几种可能。首先的一种最坏不过的可能是：人们轻易相信任务定义已经最终定稿了——如果是这样，那么他们就根本不会发现其中的错误。其次，可能会对初步的定义进行修改——虽然在多数情况下，这些修改都是非正式的，但是这些修改必然要占用本来计划用于“系统分析”的时间和精力。项目就这样继续下去，一类工作接着另一类，其间没有任何回溯过程——至少，对于项目经理来说确实如此。在项目中，已经烙上了这些经理关于程序开发项

目实施模式的烙印；而在经理的头脑中，根本没有意识到事情实际上并非如此简单。也许这可以成为一个规整的统计系统；但是，任何项目的实施都是一个包含了潜在的成功与失败的心理过程，这种系统对于理解这个心理过程没有任何益处。

即使可以通过强行的办法，把一个项目生硬地划分成若干阶段，这也绝对不是一个好主意。参与项目的程序员们的头脑中蕴藏着各种各样的才能，就特定的某类工作而言，其中的一些人可能比其他人更为擅长。如果任何时候只进行一类工作，那么总会有一部分人的才能被闲置起来。此外，如果把整个工作刻意地划分为若干部分，从而保证这些不同部分在任何时候都分别属于不同阶段，那么即使是对那些独自工作的程序员来说，这也不失为一个好主意。为什么呢？因为程序开发的环境或者该程序员自身的气质，在不同的阶段会有一定的偏差波动；如果采用这种办法，工作的进展受到上述波动的影响就会更少，从而将会变得更加稳定一致。

以编码工作为例。我们知道，并非每一天都是编码的良辰吉日。如果我们正在编码，但是却感到自己的脑子昏昏沉沉、不在状态——每向前进三步，就要接着往后退四步——那么把编码工作暂时放在一边，然后转而进行那些需要其他一些技能或者另一种心境的工作（比如文档整理），也不失为一个好主意。但是反过来，要是此时惟一可以做的工作只有编码，那么你逃避编码工作的方法也只有一种：把整个程序扔到一边。把整个工作这样地搁置起来——至少，**被别人看见**自己把整个工作搁置起来——是我们很难承受的。于是，尽管我们对自己的状态再了解不过，也不得不硬着头皮继续编码。

程序员所处的外部环境，也并非每天都是一成不变的。比如，要是因为安装一些新的设备或者检修硬件的故障，计算机需要关闭，那么上机测试也必须中断。而如果偏偏在这段时间里除了上机测试之外没有任何其他事情可做，那么整个进程就必须中断——这将会令程序员非常恼火。反过来讲，要是能够同时安排多项工作，那么在这种情况时，我们就可以转向其他工作。

这种“锁步式”（lock-step）程序开发方法，还会带来另一个问题：在开发过程中的任何时刻，一方面是一些设备处于超负荷运转的状态，而同

时在另一方面，其他的（大部分）设备却被闲置一旁、无人问津。在程序测试阶段，对上机机时的需求达到最大。要是所有人都同时进行程序测试，那么计算机必然会超载。但是那些秘书人员，在测试阶段却只能在一旁无所事事、游手好闲。反过来，在进入文档整理阶段之后，由于所有人的文档化工作是同时启动的，所以同样是这些秘书人员，其承担的任务量却会陡然达到最大，他们将会淹没在键盘的敲击声中，忙得不亦乐乎。在程序开发过程中，正是由于任何人都只能是某些特定阶段的专家，所以他们必然会一会儿闲得发慌，一会儿又会忙得不可开交。

因此，一个好的项目规划，应该首先避免将其中的各部分任务同时集中在项目的同一阶段内。但是，如果对有关管理的经典文献深信不疑的话，那么你恰恰会得到相反的印象。可以解决（而不是引起）问题的方法，是把各阶段的实际界限弄得模糊一些。这样，在问题定义的阶段，就已经可以着手开始进行一点调试工作；反过来，即使进入了调试阶段，仍然可以进行部分的问题定义工作。这种“托尾效应”（smearing）式的方法，在优秀的程序员中已经被普遍采用——无论是有意地还是无意地——这样，一旦在某个领域的工作中碰到什么障碍，他们就可以先转往另一个方向。而实际上，程序员水平低的一个标志就是，一旦由于计算机出现故障而无法运行程序时，他只会呆坐在自己的座位上，无所事事。如果不能依照预定计划进行当天的工作，他就会茫然不知所措。

很自然地，并非在每件事情上所有人都同样擅长，所以我们总是希望能从事自己最拿手的工作。比如，人们大都缺乏用平实的英语进行写作的能力；毫无疑问，这就是编写文档的工作普遍地成为人们心头创痛的原因。纵然没有其他事情好做，编写文档也决不是一项可以随意替代的工作。确实，我们可以精心地安排项目中的各项工作，以保证每个人都能被安排在自己最擅长的岗位上。但是，这种方法至少有两个缺点：首先，我们可能找不到人来做文档化的工作；其次，每个人通过项目所学到的东西甚少。

反过来，为了使程序员通过项目收获最大，我们可以把他们指定到其**不是最拿手**的岗位。同样地，通过这种办法，我们可以确保在遇到障碍时，他们会欣然接受转向其他任务的机会。

在“程序开发”的招牌下，是由各种不同类型的工作组成的多样性。这种多样性，也是程序开发的心理学研究被夸大的原因。如果我们对一个小型程序开发项目的绩效进行测试，其结果很可能会主要取决于整个工作的某一个阶段的绩效。比如在一个强调留意细节的问题中，有些程序员可能会比其他人显得更为突出。但是一旦问题要求程序员具有暂时停止手头的具体工作、对情况进行整体分析的能力时，形势就可能发生逆转。因此，如果我们对两方面都进行了研究，可能就会发现它们的绩效相差之悬殊，可以达到 30：1。

反过来，如果我们让同样这些程序员去解决更大规模的问题，我们就会发现，上述极端的差异将缩小。其原因在于，这时每一类的程序员都可能找到某方面适合于或者不适于自己的工作。对于那些达到与我们在前一节所介绍的规模的项目，我们就可能会发现其绩效差距大至相当于 2：1 或 3：1。此外在许多情况下，即使是这些差异，也不再是由于程序员之间的差异造成的，而是由于某些来自外部的因素。并非是程序员之间的差异消失了，而是由于在项目的规模增大之后，会对所有类型的态度和技能都提出要求，于是程序员之间的差异被平均掉了。

尽管如此，只要我们善于探究，就可以透过 30：1 的比例发现其背后的道理。这个比例也许恰恰体现出，**在程序开发的不同阶段**，程序员之间的能力差异。因此，只要能够把所有人的努力划分为不同类型的工作，而不是不同的程序，那么任何项目的开发效率都可以得到这种量级的提高。请注意，这恰恰就是无私式程序开发方法所能达到的效果；同时，只要程序员“拥有的”是程序，而不是程序开发的不同阶段，那么这种提高的潜在可能就无从谈起。

根据这种推断，让我们跨越整个开发周期，再回过头来看看那种企图把程序员们的工作进行真正划分的想法。在按照经典方法划分出的不同部分之间，界线是模糊的。即使能够认识到这一点，我们还是能够按照自己的方法进行进一步的细分，而且这种更细的划分会带来很大的成果。比如，让我们来考虑测试阶段。只需一眼，我们就可以看出有 3 种（从心理的角度来看）明显的可分割出来的工作，所有工作都可以归入下面这三种类别：

1. 检测错误；

2. 在已经检测出存在某个错误后，继续确定其位置；

3. 在发现错误所在位置之后，修正它。

从更广的意义上讲，我们可以发现，这 3 类工作各自所需要的技术和个性特点不尽相同。为了检测错误，程序员需要有善于怀疑的头脑，凭借这种头脑，才有可能在被认为完美无缺之处把纰漏挖掘出来。也许，捎带一点偏执狂——一种在任何时候总是从最坏的设想开始思考的方式——反而更加有益于这类工作。

但是在确定错误的位置时，我们所需要的程序员，应该能够像老妇人一般的细致与坚持不懈，并能够像林鼠一样具备收集的本能。比如在某个项目中，人们早就知道在操作系统中有一个错误，但是足足经过了 6 个月，这个错误的位置才最后被确定；而为了完成这个任务，有一位程序员把此间的所有内存转储文件都给保存了下来并进行了分析——这些文件堆成了三个各有九英尺高的小山。每天晚上，人们总能见到他面对这些文件凝神苦思——搜寻、摸索、翻箱倒柜，捕捉任何一点有可能逃过夏洛克·福尔摩斯（Sherlock Holmes）眼睛的蛛丝马迹。就这样一直过了好几个月，直到在察看某个专储文件时，他突然发现有一个比特位显得有点异常。于是他返回来重新检查所有的专储文件，又发现了另外 2 个这样的地方；不久，这个数目上升到 5 个……，直到在成百个文件中找出了 12 个这样的问题。他把这 12 个数据排列成一张很大的表格，通过对一个接一个的模式进行比较，其中的一些线索终于呈现了出来。

到这时，他对这些数据实例已经了如指掌，但是他很清楚，要是这时就向别人解释自己的假设时机还没有成熟。依他看来，似乎在某种特定的条件下，有一个数据通道会从自己的地址寄存器中挑选出一个比特位，于是就会使某个字符不能按照其正常的顺序存储。鉴于这个错误通常发生在一个大型数据块之中，这个错位的字符常常会被后续的字符屏蔽掉；但是在很少的——被他发现的 12 种——情况下，这个字符将能够幸存下来，直到数据块整体被取消。他试图去说服工程师，但是他们根本不听。程序员经常使用的一个借口，就是计算机自身的错误。所以直到最后，他不得不自己编写了一个程序，用更高的频率把通道的环境强制地设置为适宜的条件。就这样，由于他可以让错误更加有规律地出现，所以可以更为精确

地定义出错的条件，并且准确地示意出，到底是哪个电路板出了问题。工程师们终于为他的勤勉精神所感动，答应让他检查一下这块电路板——最后发现，他的推断一点没错。

虽然这类发现完全可以与 Pasteur[①] 和 Salk[②] 的发现相媲美，但是其发现者却没有因此得到相应的奖赏。尽管如此，他们还是对自己的工作非常满意、乐此不疲——这种心态已经超出了常人所能理解的地步。但是，一旦错误发现后需要进行更正时，这类发现问题的天才就会显得力不从心。实际上，问题一旦被发现，这类人就会立即失去兴趣。如果强迫他们去修正错误，那么常常会出现的情况就是：他们会在原来错误的地方造成一个更为低级的错误，或者他们会写一个笨拙而且低效的补丁程序。而为了在原来的程序上做一个漂亮的修正，最好还是请另外的一些程序员来帮忙。

如果要更正的是别人的程序，那么修正者还需要具备恰如其分的感觉，以及对计算机或语言的某种技巧有充分的了解。换而言之，所谓的“打补丁者”，其头脑必须十分“全面”；而“错误探寻者”则截然相反，他们只强调必须拥有一个“善于分析”的头脑。虽然不排除有某些人**能够**同时胜任这两方面的工作，但是更常见的情况是：一支团队所完成的工作，要比其中任何一名成员更为出色——如果他们对自己的才能与缺点有自知之明的话。

小结

程序开发并不是一个一成不变、不可分割的过程。软件设计师通常会忽视这样一个事实：与业余程序员相比，专业程序员的术业有专攻，其需要的工具也因人而异。主管们也常常会忽略另一个事实：程序开发

① L. Pasteur，巴斯德（1822－1895）。法国化学家、微生物学家、近代生物学的奠基人，发现了对于蚕病、鸡瘟、炭疽病的治疗方法，发明了巴斯德灭菌法，晚年在狂犬病疫苗的研究上作出了贡献。——译者注

② J. E. Salk，索尔克（1914－　）。美国细菌学家，1954 年首先培育出有效对抗小儿麻痹症的杀菌疫苗。——译者注

工作可以被划分为若干阶段，但是这种划分并不是整齐划一的。而程序员自己也常常会忽视另一个事实：他们的工作，只是就不同程序的不同方面来进行的。

程序开发过程天然所具备的多样化特点，会给软件设计师、主管和程序员的工作造成混淆，也使其更为复杂。但是至关重要的是，对我们就程序开发的心理学研究来说，程序开发过程的这种特点也会增加问题的混淆性与复杂性。我们已经知道，无法用任何绝对的词汇来定义一个“优秀的程序”；与此相类似，我们也无法这样来定义一位“优秀的程序员”、“优秀的程序开发主管”，以及一个“优秀的软件”。总而言之，要想提升关于程序开发的讨论的层次，我们就必须把关注和谈论的问题降到底层；如果我们能够着手对更为细致的问题进行研究，那么与通常的诸如“优秀的程序员应该具备……”等泛泛的讨论相比，我们将能够得出更具一般合理性的结论。也许我们会发现，所谓“优秀程序员的标准是什么”的问题，也类似于诸如“深厚友谊的标准是什么”等问题——实质上，它们都是人与人之间互相承认与鼓励的问题。

思考题

请主管回答：

1. 你是否曾经参加过某门“行政人员”程序开发培训课程？描述一下你当时的所作所为，以及它们与专业程序员的工作有何联系。如果在课程中曾经有某项活动，对你就程序开发过程本性的认识产生了误导，那么也请描述一下。

2. 在分配程序开发任务时，你如何估算任务的复杂程度？是通过对照任务说明进行检查？是通过倾听承担任务的程序员的反映？还是通过某些可靠信息得出的结论？

3. 就下列因素的相对重要性而言，你在布置任务时的明确程度如何？

a. 日程计划
b. 规格说明
c. 运行速度
d. 空间消耗量
e. 文档
f. 其他因素

4. 请介绍一下你目前正在负责的项目中的工作任务序列。实际的工作进行是否符合最初的计划？你是否希望这种状况持续下去？

5. 你提交的进展报告，与程序开发的实际情况在多大程度上一致？为了确认是否与实际情况一致，你作过哪些检查？通过勒令程序员严格按照预定计划行事，而不允许其根据当前工作作相应调整，你的计划是否在实际工作中得到了落实？

6. 作为一名主管，你将采取哪些步骤，以减轻关键部门所承担的工作负荷？

请程序员回答：

1. 你是不是一名专业程序员？你凭借什么成为一名专业程序员？

2. 请给出一个清单，列举出有哪些工作在 15 年前是必须自己亲自动手完成，而现在可以用现成的软件来完成。如果不能，你是否不了解 15 年前的软件是什么模样？你是否认为，作为一名专业人员，应该对本专业的发展历史有所了解？

3. 你对各类程序开发工作的兴趣与投入是否平均？你是否只偏好自己最拿手的？对于那些还不甚擅长的或者不甚喜欢的领域，你正在通过何种努力进行提高？或者，通过哪些努力使自己最终认识到：在自己最擅长的领域工作的机会与可能最大？

4. 你目前所参与的项目，其目标是什么？请根据其重要性，给出一份所有目标的清单，然后向你的主管了解一下，看看如果不看你的清单，他是否会给出一份类似的清单。对两份清单做一比较，并就其差别

做一报告。

5. 如果你的计算机需要关闭一天或者几天，以至于无法进行任何调试，那么你将如何打发这段时光？

6. 下列工作中，哪项是你最拿手的？

a. 检测输出中的细微错误；

b. 在错误被确定存在后，确定其在代码中的位置；

c. 选择简单而且有效的修正方案。

在另外两个不甚擅长的方面，你将采用什么办法加以克服？

参考资料

[] Rosen, S., ed., *Programming Systems and Languages*, New York, McGraw-Hill, 1967.

Rosen收集整理了有关计算机软件（尤其是语言与编译器）方面的大多数重要的文献，并且加上了注解。那些刚刚入行的专业程序员可以参考一下这本集子，并透过它了解一下，自己所从事的这个行业之所以能够发展到今天的水平，其间经过了哪些变化。

[] Sammet, Jean E., *Programming Languages: History and Fundamentals*, Englewood Cliffs, N. J., Prentice-Hall, 1969.

与Rosen收集古老文献的做法不同，Sammet将各类工作从达到小整合到一种统一的模式之中，并通过这种方法试图呈现出整个历史的风貌。她的这种方法之所以极具价值，其原因有二。(1) 通过把模式的概念引入到她的这本集子中，她成功地揭示了程序开发工作的模式实际上是何等的复杂。(2) 该方法的第二点价值与前一点相关，因为Rosen的集子只包含了最出色的、最成功的工作，这样的做法很容易给读者一个错觉，使他们看不到在这些成功背后的大量实验。在软件开发的过程中，主要的还是那些尝试性的，甚至是错误的实验，而且实际上错误的实验要多得多。Sammet成功地品尝到了在这些实验中所蕴含的天然滋味。当然，她也没有能够覆盖所有的错误，尤其是那些（幸亏）从未公开的。我记得有这样一次名为MYSTIC的错误。如果出于教学上的目的，需要一个令人恐怖的实例的话，那么这个例子实在是再合适不过了。

[] Metzger, Phillip W., *Programming Project Management Guide*, IBM Federal Systems Division, April 1970.

[] Lecht, Charles P., *The Management of Computer Programming Projects*, American

Management Association, Inc., 1967.

这两份文件，恰如其分地阐述了程序开发项目管理中的管理原则。Metzger的那本书是为IBM内部的程序开发项目管理方法课程而编写的导论，这门课的发起人是Al Pietrasanta、Joel Aron和一位最初的指导者。Lecht的那份文件，是全美管理协会在这一方面的贡献。这两份文件中都提供了有用的信息，还有一些振聋发聩的精辟论述。其居于上风的姿态，在Metzger那本书最后一页的这句话中，已经跃然纸上：

"你们知道，如果希望从事管理之外的工作，你不应该为这种想法而感到羞耻。"

你很可能会猜测，接下来的一句话应该是："为什么我那些最要好的朋友都不是主管？"确实，在关于程序开发的看法方面，通常的项目管理人员与程序员个体及其负责的任务之间大相径庭；但是，只要注意到在主管们——或者更准确地说，那些正在为主管们备课的教员们——的头脑中存在着的上述潜在的精英意识，那么对与这种看法相左的现象，还有谁会觉得难以理解呢？

□ Williams, R. J., *Biochemical Individuality*, New York, Wiley, 1956.

Williams从自然生理的层面出发，对人的个性进行了讨论，这本书本身也极具个性。对于那些深受"所有程序员半斤八两"这句话之苦的人们来说，这本书不可不读。有些人有三个肾，但是另一些人只有一个；要是我们能够理解到这一点，那么为什么还会轻信：为了完成同一个任务，所有程序员花费相同的时间？

□ Tyler, L. E., *The Psychology of Human Differences*, 3rd ed., New York, Appleton-Century-Crofts, 1965.

Tyler所讨论的问题，与Williams完全一致；不同的是，他是在心理的层面，而不是在生理层面上进行讨论。这两本书都出现在1956年，而FORTRAN语言也诞生于这一年，难道这是一种巧合？

第 7 章　程序开发任务的差异

评　　注

要是在今天来写这本书，也许我就根本不会想到写这一章了，因为时至今日，程序开发任务的多样性是如此之明显，以至于任何人都已经注意到了这个现象。

在为本书的第 1 版进行准备时，Ed Schulman 和我曾经把在本章中介绍的那些实验加以扩充，用以描述除有效性程序开发和快速程序开发外的其他方面[①]。纵观历史，可以毫不谦虚地说，它们是我在程序开发心理学方面所见到过的最为至关重要的实验。我们通过实验所揭示的是：无论被指定**任何一项任务**，程序员**几乎都能够完成**。（然而这并不等于说，无论你给他们指定**多少任务**，程序员都可以**全部**完成——尽管有些主管的确这么认为。）

受到这些实验的影响，我认为我所投入时间的最大回报，就在于发现了那些明确的需求，并且进行了相应的沟通。而且，这个题目的确很慷慨地让那些对此关注的单位得到了回报。

① 有关这些实验的报告请参阅：G. M. Weinberg and E. L. Schulman, "*Goals and Performance in Computer Programming*", Human Factors, Vol. 16, No. 1 (1974), pp. 70—77.

第8章 人格因素

从某种意义上讲，“**人格**”这个概念包容了我们可以在一个人身上发现的所有个体差异。即使是对人类行为最不经意的观察者，也可以明显地看出，人的智力及其受到的教育会影响到人的人格。从上面一句话的意义上讲，当我们使用“人格”这个术语时，我们所指的是我们通常用来刻画一个人时所使用的所有性格特点的总和——也就是对一个人的某种总体介绍。从这个意义上讲，一个人的人格就是其有别于他人的特点；在一个计算中心所进行的一项研究，清晰地阐释了这一概念。

疯狂的轰炸机

在一个远程批处理的环境中，由机房中的操作员负责运行由远程的程序员编写的程序，但是在这些操作员与对应的程序员之间，即使有直接的交流，也只是凤毛麟角。在这样的情况下，我们可能会很自然地猜测：操作员应该不会对程序员的人格有什么了解。但是，实际上的情况并非如此。作为我们的个人特征所有方面的总和，人格必然会体现在我们的一言一行之中。尤其是，人格将会在我们所编写的程序中得到体现；而且更特殊地，它将体现在计算机操作员所看到的那些程序之中。

由于其特定的工作性质，其中的首席系统程序员要同时与程序员和操作员做面对面的交流。因此，他发现操作员经常会对某些程序员大肆诋毁。在参加我们的“程序开发心理学”讨论会期间，他决定对操作员的这些评论加以研究，并期望通过努力将这些负面的评论与特定的程序开发工

作关联起来。但是，他的一项最有意思的研究结果，竟然与两名程序员的人格有关。

在使用该计算机的所有 70 名左右的程序员中，操作员们对这两位程序员的评价最为突出。一般来说，不同班次的操作员不会有机会有互相讨论的机会；但是即使是属于不同班次的操作员，也都一致认定：这两名程序员是所有人当中最差劲的。在做了进一步的调查之后，这位首席系统程序员发现，大家之所以得出这样一致的结论，都是因为这两人的程序所引发的异常程序中断——造成内存转储并且“摧毁”整个系统——有关。除了这两人的程序之外，这种异常程序中断极少出现，因此，操作员对他们最为恼火。但是，当时并没有合适的渠道，可以使操作员与这些程序员进行沟通，所以一旦整个系统被“摧毁”了，他们惟一能做的只有不断地咒骂。

系统程序员于是开始收集这种灾难性事件发生情况的样本。最后，他收集到了足够多的样本，并通过分析确定了明确的事件发生条件。于是，他把这两名程序员找来面谈。他发现，第一位程序员虽然是一名工程师，但是从来没有接受过程序开发方面的任何培训。但是，他所在的项目小组的工作却非常依赖于来自某一个程序的结果。而自从该程序最初的作者离开后，就一直由该工程师负责维护。由于对程序开发一窍不通，同时又耻于向别人寻求帮助，所以他不得不通过反复尝试来运行该程序，一会儿试试这块卡，一会儿又试试另一块卡，直到最后获得自己所希望的结果。尽管在这个过程中，他不断地让操作员大跌眼镜，但是很自然地，他根本没有意识到给别人带来了何等的麻烦。

第二位程序员捅的娄子更大，被操作员们“亲切地”称呼为“疯狂的轰炸机”。但是实际上，原来他的情况满不是那么回事儿。他对程序开发具有丰富的经验——实际上，其经验长达 15 年之久。根据他自己的介绍，在这 15 年期间，他曾经在 11 种计算机上、使用过 14 种语言编过程序。而且同样是根据他的介绍，他在所有这些计算机和语言方面都堪称专家。正因为如此，他的程序会摧毁计算中心的操作系统就没有什么奇怪的：这些程序过于深奥，以至于紧紧依靠操作员是无法弄懂的；而且，在一个不是经他本人亲手编写的操作系统上，根本不可能处理这些程序。

这些未经修改的程序被反复地提交，正是这种凑合的心态，使这些出

自专家之手的程序导致系统崩溃的事件明显地增加了至少3倍。据他自己认为：他的程序出错的可能性应该很小，所以除非在一行之内被拒绝了三次，否则根本不值得浪费他宝贵的时间，去看看程序中可能导致问题的地方。在这种情况发生时，他也不过是踱到其他程序员的办公室，命令式地询问一下，系统为何工作异常。

即使是在别人耐心地指出其错误所在后，系统崩溃的事件依然没有杜绝。系统程序员就发现过这样的一个例子：这个程序由三个部分组成，但是这架“轰炸机”在第一部分中的任务控制部分犯了一个错误。在被告知错误所在以及如何进行修改之后，他重新提交的程序再一次摧毁了整个操作系统。在一次又一次这样的灾难之后，人们又指出其在第二部分中也犯了一个错误！这还没有完。正如我们会猜到的，当第三部分运行起来后，其中还是暴露出来了同样的错误。

这两个人的人格有天壤之别：一个像老鼠，另一个则像狮子。尽管如此，他们都是说明人格对程序开发绩效产生影响的最典型实例。只要让那位工程师去参加一门有关程序开发的课程，就可以使他获得原来自己碍于脸面难于获得的帮助，于是他所引起的灾难也就很容易消除掉了。但是对于那位“专家”来说，培训并不能解决问题。原因就在于他过于顽固地自信，认为自己对程序开发的任何方面——或者，至少任何值得了解的方面——都了如指掌。所以，根本没有办法说服他去参加什么课程，甚至连其他人的任何建议或忠告，他也听不进去。为了“解决”问题，最后只好请他到别的计算中心另谋高就；而且正如众所周知的，这类“疯狂的轰炸机”直到今天依然存在。

人格的改变

尽管没有为众人普遍接受的关于人格的定义，在我们关于人格的研究中，可以把下面的定义[①]当做一个行之有效的模型——因为根据这种定义，

① D. Krech, R. S. Crutchfield and N. Livson, *Elements of Psycholoy*, 2nd ed., NewYork, Knopt, 1969.

人格既会影响程序开发，反过来也会受到程序开发的影响：

> 所谓人格，就是一个人的所有特征的集合；根据这种具有惟一性的集合，能够确定这个人如何适应不断变化中的环境，同时反过来，这个集合本身也会受到这个适应过程的影响。

这个定义多少会让有些读者感到吃惊，首先的一点就是，该定义居然声称，一个人的人格并非如通常所说的“江山易改，秉性难移”。的确，在人格中确实有些部分是经久不变的，所以尽管有时从表面看起来似乎有所变化，但是除非有特殊的原因，否则确实是“秉性难移”。比如，一位相对看来和蔼可亲的程序员，可能会开始变得不愿与同事们交流，并且对别人在工作中提供的帮助做出情绪暴躁的反应。其行为上的这种变化，通常发展得很缓慢，也许要经过很长的时间才能被别人察觉。实际上，造成这种问题的条件有时只是短暂的一会儿；而等我们意识到这是一种“人格的变化”的时候，这种行为上的变化早已恢复回去了。但是，一旦出现永久性的变化，那么其背后就必然存在某种原因；尽管从任何角度来看，这种原因都很可能似乎与实际的行为无关。

在我们的实验中，主管是通过来自其他同事的抱怨，才开始注意到这位原先态度和蔼的程序员的人格变化。在一次交谈中，该程序员承认，他也注意到了自己在行为举止上的变化，但就是找不出原因何在。主管也曾探究过其原因，但是也和这位程序员一样无功而返。但是，经过几个星期之后，这位程序员就提出要提前告假，原因是他实在受不了自己牙疼的折磨。实际上，虽然他曾经想过去看牙医，但是这几年来一直没有去成。这次，他终于下定决心去看牙医。但是牙医检查过后告诉他，有四颗牙已经感染；由于感染非常严重，这四颗牙都必须立即拔除。拔牙之后才不到一个星期，他从前温文尔雅的形象又重新出现在人们面前——原来，他此前性格变坏的原因，就是来自于牙齿的感染。

不幸的是，人们往往不能在事先确定人格变化的原因，一般要等到事后很久，才能搞清楚到底是为什么。通过这件事，我们并不能简单地得出这样的结论：一旦某个对人友好的人变得待人冷漠，那么就需要拔掉他的几颗牙齿。人格在同一方面的改变，可能同时源自于多个方面的原因——身体的、精神的，甚至与当事人本人完全无关。有另外的一个例子：在一

所程序开发学校里，有一个人表现得很糟糕，但是他所属的办公室此前为他写的推荐信却对他大加肯定。学校的主任虽然已经准备让他不通过并且卷铺盖走人，但还是不厌其烦地和这个人一起吃了顿午餐，以给他一个机会为自己做个解释。还没有等到上甜点，这位成绩差的学生就把其中的原委和盘托出：他独自在1 500英里之外的妻子，刚刚死于癌症。能够来这所学校进修，对于他的前程发展来说是一个重要的机会，所以他的妻子曾坚持让他来这里。但是，尽管他已经竭尽全力去实现妻子的愿望，而他就是无法把精力集中到学习上。由于不想因此而受到别人的特殊关照，他一直没有把自己在校内的处境告诉任何人。虽然因为妻子的去世而痛不欲生并不是什么过错，但是如果这位主任没有这样的耐心，恐怕这位程序员就要带着耻辱去见江东父老了。

上述的两个例子，可能会让任何一位主管陷于道德上进退两难的境地；至少，在我们这样一个认为不应该把“私生活”带到办公室来或向自己的主管和盘托出的社会里，这类问题确实很难解决。但是，仅仅根据症状不可能确定，问题的症结到底是来自办公室内部还是外部；因此，主管们没有别的选择，他们只有努力去理解那些与手头的工作似乎不相干的行为。如果问题的根源来自办公室内部，那么就很容易加以纠正。也许某个程序员与办公室里的某个同事合不来，但是只要不是和每一位同事都合不来，通过调换办公室就可以很快地解决“人格”问题。

问题并非总能如此轻而易举地得到解决。尽管如此，尽可能地进行一些调整，至少总能多少缓和一点问题。例如，如果某个程序员对计算机操作员的态度变得有点暴躁，那么很可能是由于他承受的压力过大——比如他需要完成一个棘手的程序，但是又无法确定其中错误出现的位置。而要是该程序员偏偏又过于自负，那么他就绝对不会承认自己遇到了困难，更不用说去寻求帮助了。但是，生硬地禁止该程序员进入机房只能治标，不能治本；用不了多久，他的问题必然又会在其他方面表现出来。反过来，要是问题的原因已经找到，那么接下来就可以给他提供帮助，因为这时他不用承认自己遇到了困难。虽然对其程序开发能力的质疑会使他恼羞成怒，但是一旦错误已经被找到，他很快就会把它忘到九霄云外。我们都会时常感到，用一剂猛药来治病，其后果可能会比疾病本身还要糟糕，而它

至少会很快就会失效。

在任何情况下，如果没有首先挖掘出问题的根源，那么针对人格问题进行校正或调整的有效方案根本就无从谈起。在确定此类问题的症结所在时，虽然懂一些心理学的知识将很有帮助，但是目前的心理学还远没有发展到抛开当事人就可以诊断出问题的水平，而且也永远不会发展到这种水平。如果只是简单地对一个人的工作进行观察，那么你绝对不可能看出其妻子得了癌症、他的牙齿有毛病或者他对自己不能按时完成任务感到恐惧等问题。如果一位主管只是根据其外在的症状来做出心理方面的判断，那么他永远不可能获得成功。但是，在发现这些外在症状之后，如果这位主管不是急于采取措施，而是根据这些线索继续深入了解情况，那么他最终采取的措施就非常有可能获得成功；而这些更为深入的信息，只能从当事人那里获得。

人格中经久不变的部分

既然一个人的人格“会在适应环境的过程中不断变化”，那么我们就可以通过人们在这一过程中的人格变化，来了解其所处环境的变化。但是，只有在环境出现极端变化之后，人格才有可能发生变化；而且，人格的变化对任何人来说都不是家常便饭。而我们应该更加关注的，应该是人格中或多或少的那些经久不变的部分；因为，这些部分对程序员的影响更大。

关于人格的一种提法是：人格就是人的各个**人格特征**的集合。从心理学家到凡夫俗子，这个提法都被普遍地接受。有的人可能会被评价为“保守、坚定而精明”，而另一些人则可能被评价为“腼腆而富有想像力”。这种描述一个人的方式确实不错，我们甚至会依照这种模式，来列举出“理想程序员的人格类型”中所包含的人格特点。但是在“程序开发”这名词之中，同时包含着千变万化的情况；有鉴于此，我们顶多只能奢望，在某些人格特征与某些类型的程序开发工作之间，能够找到一些联系。

我们不必去讨论各种的“人格特征理论”，因为虽然这些理论可以为我们论述的严密性提供保证，但是我们在这里并不需要很严密。另一方

面，虽然有些理论可能尚无定论或者并不精确，但是这并不妨碍我们从中获取有益的启示。例如，“信赖他人”和“多疑”是一对被经常提及的、相互矛盾的人格特征；同时，我们也会时常扪心自问：作为一名优秀的程序员，到底应该具备其中的哪种人格特点呢？如果一位程序员正在进行调试，那么很显然他应该更加多疑才好，如果有点偏执狂就更妙了。但是反过来，在需要与其他同事合作的场合，大家之间就需要更多的信任，因为如果你不信任别人，别人也不会信任你。但是在有些时候，这两种人格特点会发生冲突。如果有一个程序员所在的小组负责“产品测试”，那么他就**必须**对其他人的程序持怀疑的态度。要是有人对他说：“不必麻烦你检查了。我已经把程序修补好了，并且自己做过了测试。”那么，作为一个乐于相信别人的人，他就不应该对此人的话有任何怀疑；但是，他现在正在承担的任务却恰恰需要他对此保持怀疑态度。

于是，这位产品测试员就处于一种进退两难的境地。但是，他实际上会在多大程度上感到左右为难，最终还要取决于他在其他方面的人格特点。如果此时此刻他特别盼望得到众人的喜爱，那么要是看起来他不得不对别人的话提出疑问，那么他就会因为这种似乎不太友好的举动感到为难。反过来，要是他更具有我行我素的性格特点，那么尽管同样会让其他人感觉他怀有敌意，但他还是会毫不犹豫地怀疑其他人关于他们自己程序的陈述。这样，如果其他方面完全一样，那么有些人就会在心理上觉得产品测试的工作异常轻松。

但是，和通常的情况一样，所有其他的方面不可能完全一样。比如，在人格上可能会有情绪稳定和不稳定之分，这方面的不同可以决定一个人能够在尴尬的处境中坚持多久。由于时间的长短很重要，所以虽然由于其工作性质决定产品测试员必须具备一种冷峻的性格，主管还是可以通过更加频繁地调换手下人的工作，以缓和产品测试员在这方面所承受的压力。通过采用无私式程序开发的方法，这种任务角色的轮换可以自动实现，因为在这样的环境中，任何人的人格都不会受到冷酷无情的冲击——这种冲击如果存在，就会迫使人们的人格做“适应性调整”。

这样，通过用无私式程序开发方法，我们牺牲了**总是**把每个人放在最适合其人格的岗位上的可能性；但是正是通过这种方法，安全性和稳定性

等方面的一些潜在效率却可能得以实现。此外，由于我们并不是把人格的完美组合视为头等重要的方面，所以并不会因此而损害到这种效率的完整潜力。同样地，培养容忍宽容人格的最佳方法，就是“去穿穿别人的拖鞋。”

如果你试图为每个人找到其最适宜的岗位，那么就可能在另一个方面遭受失败。实际上，一个人的人格并非只是简单的一层，而是包含了丰富的层次。即使是相同的层面，其中隐藏的内容也可能很不一样——甚至截然相反。一个人之所以会从表面上看起来“待人友善”，可能是为了保住自己不安定的地位，故意给众人制造一个良好的印象；当然，也可能是他在自己的位置上泰然处之，伴随着心情的放松，他性格中友好的一面得到了彻底的显示。显而易见地，我们必须特别谨慎，不要仅仅凭借表面的简单现象来推断一个人内在的人格。

对于其上级、同事或者下属，许多人表现出来的态度是不尽相同的；这类因素，也可能使我们对某个人的人格做出错误的判断。另外，人们在不同场合中的行为之所以差异很大，还有诸如性别、年龄等其他方面的原因。有这样一个故事可以形象地说明这个问题：有一天直到夜里很晚，某家计算机制造公司的一支程序开发团队仍然在测试室对一种新型计算机进行测试。在凌晨两点时，工厂的一名保安发现他们还在紧张地工作着。这名保安对此非常不乐意，于是带着傲慢的口气向众人指出：这个工厂禁止女性工作到午夜之后。其实，其中的那位女性正是这支团队的主管，另外还有三位年轻人。于是，其中一位年轻人站出来为主管辩解：

“你的意思是说，在这里女性根本没有可能在午夜之后继续工作?”

“是的，”保安说出的每个字里都饱含着威严。“你们统统必须立刻离开。我会把你们带到出口处。”

“但是，我们确实得到了工厂主管的许可。我们必须连夜工作，因为这台计算机在白天没来得及处理完。”

听到“工厂主管”这个关键词之后，保安的口气中开始出现了一丝迟疑：根据规定，只有在某位主管在场时，妇女才可以工作到午夜之后。接下来，他对团队的这四位成员仔细打量了一番。在确定没有一个人超过 30 岁之后，他的口气又重新生硬了起来：“但是，这里根本没有谁是主管，

不是吗?”

这时，该团队中的那位女性成员站了出来，面带迷人的笑容，张口却是一副管理者的官腔：“但是，**本人**正是……这个小组的……主管。”

保安感到十分吃惊，以至于一下子连嘴也不听使唤了。他大张着的嘴半天也没能合上，心里纳闷：难道，这样一个**女人**，这样一个**年轻的女人**，这样一个**魅力十足的**女人，竟然是位主管？当他过了半晌终于缓过味儿来，已经是另一副柔和的、略带谄媚的口气：“噢，我明白了。请您原谅，女士。很抱歉打扰您了。如果您愿意，我将每过个把小时就过来看看是否一切正常。”

女主管从他的话里听出了他的尴尬，于是转成一副管理者特有的、带有慷慨意味的腔调：“我很清楚，没有这个必要。不过，还是要谢谢你的好意。我们对你的工作深表谢意。”这话的意思很明显，保安不可能听不出来——他在这里是多余的。于是，他没有再说任何废话，就迅速地转身，然后消失在黑暗之中。要是他真的长了一条尾巴的话，他肯定会把它夹在两腿中间。

工厂的一名保安还不是程序员，所以不可能像在程序开发中居于一般地位的程序员那样，对居于权威地位的女性，年轻女性，甚至魅力十足的女性有所了解。但是，无论是程序员还是工厂保安都具有其自己的人格——正是这种人格，使他们根据自己所察觉出来的周围人的角色不同，改变其外在的行为表现。主管们可能曾经感到过奇怪：为什么有的程序员在他们看来，简直就是待人友善、善于配合的理想典型；但是在对待其同事时，却显得那么的具有攻击性、叽叽喳喳而且总是难以相处。如果明白上面讲的道理，我们就不会和这些主管有同感了。

关键的人格特征

如果说对一个人的人格进行衡量并非一件易事，那么在人格与工作要求之间做一最佳匹配，就几乎是不可能的。但是，难道我们就没有任何的方法，根据程序开发的任务来选择具有最佳人格的人吗？无论如何，既然人格因素在程序开发过程中那么重要，我们绝对不应该不经过尝试就轻易

放弃。也许，纵然我们不能就一般性的实例深入研究，但是哪怕只是通过对极端实例的考察，我们也可以得到一些启发。

人格因素对程序开发成功与否的影响作用，是极其微妙的。但是，也许关于人格特点如何导致程序开发的**失败**，有些方面我们还是可以确定的。一般的程序开发主管大都会说：对于程序开发的成功而言，人的智力因素要比人格因素更为重要。毕竟还是有很多人最终被发现其智力远不足以胜任程序开发的工作，但是尽管如此，这些主管们当中很少有人会引述这样的例子；相反，关于那些气质性地无法胜任程序员工作的人，所有人反而都会津津乐道。正是在这种意义上，我们才可以断言：在程序开发过程中，与人的智力因素相比，其人格因素才是更为重要的。

关于人格因素对导致程序开发失败的决定性作用，只要我们更为深入地研究一下，就可以很容易看出：在决定雇用程序员时，我们在其人格方面要比在其智力方面更容易犯错误。首先，通常都有大量**预先设定的选拔标准**，这样就大大限定了人们申请程序员——哪怕是作为实习生——岗位的可能。这些预定的标准中，有一部分是由明确的雇人政策所强加的，比如要求具有大学学历，甚至需要具有在某些特定领域的学位。更为重要的原因可能是自我选择的存在，那些涉世未深的人们大多相信：参加程序开发需要“大量的数学知识”。直到近几年之前，外界的人们对程序开发的认识还是扭曲的，他们总是认为：如果一个年轻人表示对计算机感兴趣，那么就应该建议他去“学习电子学”；当然，年轻的女性则想当然地被拒于计算机的门外，因为女孩子们不是做“工程师”的材料。

虽然很多人本来可以成为一名出色的程序员，但是由于这种预先设定的标准，他们往往与这个职业总是无缘。但是，无论这种错误之危害有多大，它至少还是起到了一个作用——大范围地淘汰了那些智力平平的人。要不然，如果这类人得以大量地加入到程序员的行列中来，其中的大部分都必将面临失败；这样，就会给我们造成一种感觉，认为应该对前来应聘程序员工作的人根据其智力进行筛选。但是，难道根据人格的、可比较的人才选拔方式，同样也没有吗？毫无疑问，这种方式应该存在。尽管智力本身已经不能简单地通过智商或者程序员天资测试的分值来衡量，但是和智力因素相比，一个人的人格因素选取的方面则还要多得多。因此就程序

开发的需求而言，如果依照人格作为标准来选拔程序员，那绝不可能得出统一的衡量结果。此外，即使是在学校内，根据人格对人的挑选也是与根据其智力的挑选一样严格（不要误会，在学校期间的成功同样需要有人格的基础作为保证）；尽管如此，和智力比起来，人格还是显得更容易改变。

实际上，关于人的这两个方面，心理学家们可能指出的真正可靠的性质之一就是：和人格相比，人的智力更不容易受到其所处环境的影响。比如结婚，可能对于一位大学毕业生的智力来说，不会有任何可测量出的影响；但是即使是对心理学的外行也可以轻易地看出，结婚肯定会引起一个人的人格改变。正因为如此，即使某个人在其光棍岁月里可能曾经是一位轻浮的花花公子，但是他在其今天所从事的程序开发工作所表现出来的敬业程度却仍然可能会让我们叹为观止；当然，要是他在大学期间连起码的声明语句都不会写，那么无论是结婚还是家庭，都不会对他的工作有任何帮助。

对于程序员来说，他们的哪些人格特点将会预示着将来可能的失败？鉴于到目前为止关于这个问题还没有任何的正式研究，所以我们只能就趣闻轶事来做一讨论。然而，就当今的程序开发工作而言，如果一个程序员**缺乏在高压力的环境中坚持一个多星期的能力**，也许我们就可以肯定地说，他不是一块程序员的材料。当然，我们所讨论的是专业的程序开发——其特点是，工作及其日程计划都是由外部强加的。而对于业余的程序开发来说，也许是因为其从事者的饭碗不会维系在“是否能在下周四前找到某个程序错误”之上，所以这种工作充其量只不过是一项轻松的消遣。

由于程序开发工作的多样性，所以一个人只要稍微有点**不适应快速的变化**，他就不能胜任专业程序员的工作。有的时候，程序员花费了整整一个月才完成的工作需要撤销掉，或者至少需要进行大的改动，那么他此前的心血只能付之东流。在这种情况下，程序员不可避免地会在心理上遭受重创；而要是在突然之间变成“垃圾”的是其毕生的心血，那么其心理创伤之大就更不用说了。

谈到垃圾，在程序开发工作中所需要的一个最容易确定的人格特点，就是稍微讲究一点**整洁**。我们的意思，并非指个人边幅的修剪（有这样一

个例子：有一位程序员身上的气味很难闻，以至于如果要和他坐在一起看程序，其他人捏着鼻子也坚持不了多久）。我们所讲的，是一种尽量把自己手头的资料规整好的习惯。要是大家都没有这种习惯，那么计算机的“造纸能力”最终不可避免地会令我们吃尽苦头。有这样的一个计算中心，在挑选实习生时会让他们参加一个测验，但是最终录取的原则，并不是“分数越高者越优先”，而是“其提交的试卷越整洁者越优先”。

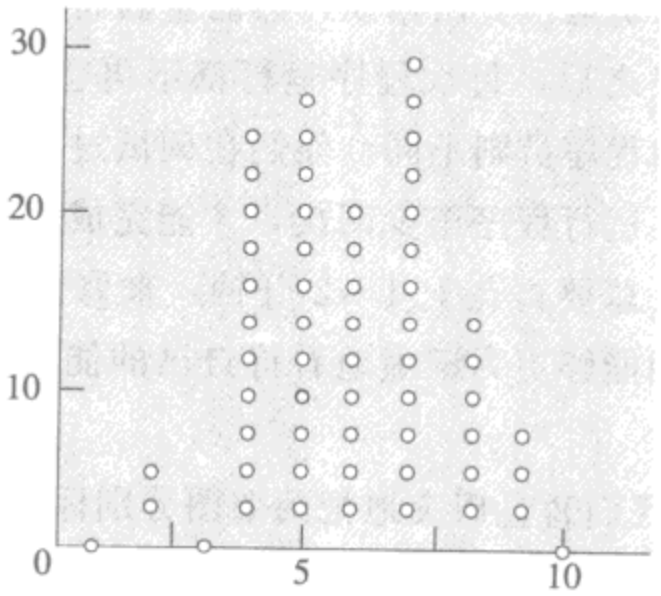
对于从事程序开发工作的人们来说，至少应该多少具备一点的另一重要人格特点就是**谦逊**。一名程序员如果不懂得谦虚，那么其命运将注定是希腊戏剧式的：成功首先导致骄傲自大，进而**狂妄**又会导致隐性的自我毁灭。即使索福克勒斯（Sophocles）[①]，也设计不出比这更好的情节，来表现程序员的自负——他们不过才刚刚学会几种简单的雕虫小技，就沾沾自喜，以专家自居；他们为自己虚构的美梦，最后总难免被计算机（程序开发过程中的解围之神）难以抗拒的强大力量碾得粉碎。

与谦逊对立的是**自负而果断**，或者称为人格力量。程序员的任务就是把事情办好，而为了办好事情，有时就必须绕过困难、跳过困难或者战胜困难。谦虚的人对于他们犯错误的可能非常敏感；其头脑中的批判意识成为他们人格力量的主要部分。缺乏批判意识的人格力量，正如没有安全阀门的**蒸汽锅炉**；但是反过来，缺乏人格力量的批判意识，也正如徒有安全阀门、但是却没有蒸汽的锅炉——虽然绝对不会有爆炸的危险，但是由于缺乏动力，工作也绝对不可能完成。

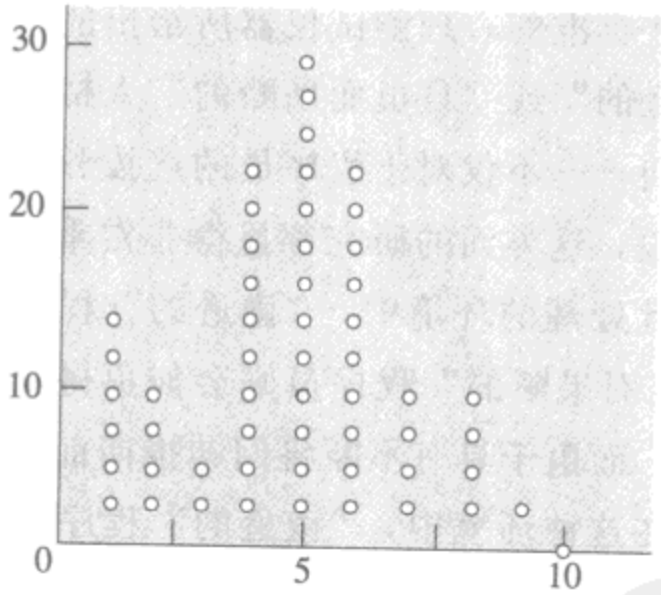
关于程序开发中谦逊与自负而果断的性格的比较，我们有一些意想不到的基于试验的证据。我们曾经设计过一个实验，以研究各种任务说明的效果。在对实验数据作进一步的细读后，我们绘制了一张图表，用来显示在对其程序在批处理系统中的运行结果进行分析时，每个程序员所花费的时间。当把他们用于分析的时间长短与分析的频率进行对照时，我们得到了两张完全不同的图表（如图 8-1 所示）。两张图都显示了频率与时间的对数之间的关系：在上面的那张图中，只有一个尖峰；而在下面的那张图

① Sophocles，古希腊三大悲剧家之一。名作有《埃阿斯》、《俄狄浦斯王》、《安提格涅》、《俄狄浦斯在科林斯》等。他善于描写命运对个人的无情摆布。英雄人物的成功—自满—覆灭是他剧作的重要主题之一。——译者注

中，除了这个尖峰之外，同时在低端还有另一个尖峰。后面的这个尖峰显示出，绝大多数的工作时段都是一分钟、两分钟或三分钟，这类时段中相应的工作都很简单，一般是纠正那些因粗心大意或指法疏忽所造成的错误。



谦逊型性格



自负而果断型性格

图 8－1　程序员的两种人格特点

注：工作时段的频率与工作时段的长短之间的关系。其中，工作时段的长短以 2 为底取了对数：1 代表 1 分钟，2 代表 2 分钟，3 代表 4 分钟，而 5 则代表 16 分钟。

但是，当我们试图在其他类型的工作中找到类似的模式时，却遇到了困难；在对 PL/I 的语法错误进行研究时，这方面的困难尤为明显。在此之前我们之所以能够看出这种模式，是因为我们只要求程序员提交其在语

法分析时出现的错误运行结果。但是，当我们收集了所有的运行实例之后，正如在对“任务说明”进行研究时一样，我们也可以看到同样的模式，只不过其形式不同罢了。在找到其最初的语法错误之后，某些程序员将**再也不会**犯同样的错误。他们在修改过程中，每次可能找出一到两个错误。他们会认真而系统地检查出错误，然后重新将卡片穿孔。所以，只要经过最初的几次修正之后，每次程序运行都不再会出现语法错误。但是，另外有大约占 1/3 的程序员则不同。他们在调试过程中总是慌慌张张、毛毛糙糙，以至于总要运行程序至少两次，才能完成一次更正。在第一次运行之后，这类程序员总要再花上 1～2 分钟，来修正那些本不应该出现的语法错误。要是他们能够更为谦虚地看待自己的能力，就根本不用白白浪费这些时间。

在图 8-1 中，我们暂且唐突地把两个图分别标为“谦逊的”和“自负而果断的”。图中的模式一目了然，而且这些模式非常稳定（我们用不同的程序对相同的程序员做了测试，他们的行为模式依然相同）。尽管如此，我们还是不能像有些标准的心理测试仪器所给出的尺度那样，简单地把这些模式归结为“谦逊的”或“自负而果断的”人格特点。显然，这是一个值得深入研究的方向——不仅对于程序员的选拔和培训，而且对于分时系统的设计与评估来说，这方面的研究都显得非常重要。通过上述结果，我们也许会猜测：在批处理的环境中，“谦逊的”程序员将表现更佳；而在分时系统中，“自负而果断的”程序员则会如鱼得水，因为他不会因为自负受到严厉的惩罚，而由于具有不断将问题推向前进的意识，他们很快就会脱颖而出——而在这种环境中，“谦逊的”程序员只能坐在终端的屏幕前望洋兴叹。

在对程序开发至关重要的各种人格因素中，最后一个应该是**幽默感**。计算机“确实让我们都成为了傻瓜”，所以，如果某个傻瓜缺乏自嘲的能力，那么在乏味的程序开发过程中他肯定坚持不了多久。一些独具慧眼的人指出：程序员的国歌就是“啊啊啊啊啊……”。每当重新焕发生命的朝气，我们总是发觉自己又一次掉进了愚蠢的假设、畸形的练习或者无知的错误的陷阱之中。只有不断地哼唱第二小节的“哈哈哈哈哈”，才能使我们长时间地忍受自己小丑般的角色。

人格测试

我们所描绘的关于程序开发的理想人格，令人印象深刻。它之所以被称为理想，是因为其中排除了可能导致失败的所有极端人格因素。这种理想的人格，并非我们通常所指的“冷酷无情的”人。鉴于程序员成天与之打交道的都是冷冰冰的计算机，所以一些世俗的观点往往会认为，程序员本身也必然是冷酷无情的。但是实际上，这种推断与实际情况恰恰相反：正因为计算机缺乏灵活性，所以程序员才更需要灵活——只有这样，计算机与人的灵活性相结合，才能如虎添翼，进而满足现实世界中的种种需求。既然如此，我们是否可以找到某些对人格进行测验的方法，从而使我们在选拔程序员或者实习生时，避免受到自己头脑中世俗观念的影响呢？

现在已经有了许多的“人格测验”方法，在帮助选拔可能的程序员时，其中的一些方法已经被偶尔采用了。这些方法本来是为了检测人格失调而设计的，而且主要用于这个方面。这些方法的例子包括：著名的罗尔沙赫氏墨迹测验（Rorschach Ink Blot Test）[①]、主题统觉测验（Thematic Apperception Test，TAT）[②]，以及明尼苏达多重人格测试表（Minnesota Multiphasic Personality Inventory，MMPI）[③] 等等。其中的明尼苏达多重人格测试表方法，是根据经验性信息而设计的。这种信息的一个获取渠道，就是通过对送交治疗的精神病人进行的研究——之所以采用这种数据，是因为经过训练有素的心理学家和精神病医生的诊断，一般认为这些人的人格特征可以很准确地得到测定。这些测验也可能对“正常”人进行，但是明尼苏达多重人格测试表方法的主攻方向只是为了协助对精神疾

① 一种视被测对墨渍图案的反应而分析其性格的实验。Herman Rorschach，瑞士精神科医生，他在1921年出版的《心理诊断学》（*Psychodiagnostic*，副标题“知觉诊断方法结果”）一书中介绍了该测验技术。——译者注

② 在一套展示日常生活的标准图片暗示下，让被试讲述一个故事的一种测试试验。C. D. Margan和H. A. Murray于1935年在《神经精神病学档案》一刊上发表了“主题统觉测验——一种研究幻想的方法”一文，并于同年将此测验应用于Harvard心理诊所，后Murray于1943年出版了《主题统觉测验》一书。——译者注

③ 由明尼苏达大学教授Hathaway和McKinley于1940年提出。MMPI是目前应用最广泛的人格测验，尤其广泛应用于临床工作和理论研究中。MMPI于20世纪80年代被引进我国。——译者注

病的诊断，而不是为了把“正常”人划分为不同的人格类型或特性。

当然，在招聘程序员时，我们要寻找的并非是精神病患者——尽管目前还没有任何来自经验的数据能够证明或者反驳这种观点。也许有人会对将这种方法应用到招聘程序员方面提出质疑，但这并不是一个什么有效性或者适当性的问题，而只不过是在招聘过程中面临的一个道德伦理问题——把针对精神疾病的测验方法应用于任何类型的工作中，这种做法是否欠妥？

最后这个问题的答案，并不只限于程序开发，所以我们暂且还是绕过其所涉及的道德伦理问题。的确，有些人格测验方法，是针对在确定工作甚至是职业时挑选人才的需要而专门设计的。那么，这些测验方法又如何呢？这类测验的一个典型例子就是斯特朗职业兴趣调查表（Strong Vocational Interest Blank，SVIB——其中，Strong 是一个男子的名字，不是一个形容词，意思并非说该测验检查的是“浓厚”的职业兴趣）[①]。在这种测验中，只有在将被试的各项“得分”与来自不同职业的人群的得分进行比较之后，才能得出最终的解释。比如，如果有一位择业顾问对一位高中学生进行这种测验，他就需要将这名学生的得分或者其对应的人格模板，与来自不同职业的被认定为“典型”的人格模板进行比较。这样，择业顾问就可以确定，哪些职业与该学生的人格模板最为吻合；一般地接下来，顾问就会从这类职业中进行挑选，向这名学生推荐一个适宜于他的职业。

当然，在进行招聘时雇主通常只对一项（或者，至多不过少数几项）工作感兴趣。通过将应聘者的情况与相应的各种人格模板进行对比，雇主就能够确定该应聘者是否具有适宜于这项工作的人格特点。那么，即使我们已经了解了人格与程序开发之间的关系，这个过程本身又到底会有多可靠呢？

这个过程的第一个弱点就在于，可能根本就没有对应于所谓“程序员”的标准人格模板。已经有很多的公司在选聘程序员时，采用了斯特朗测验；但是由于没有程序员的人格模板，它们会把程序员判断成“类似”

① 该调查表由 E. K. Strong 于 1927 年编制，是最早的职业兴趣测验。——译者注

于数学家、工程师、作家或者诸如此类。由于这类判断是建立在推测的基础上的，这些选择过程的效果与简单的掷骰子差不多。只不过，掷骰子在名义上不那么正规。在向上级主管汇报时，人事主管可能会说："我们在挑选程序员时，使用了斯特朗职业兴趣调查表。"而要是他换了另一种说法："我们在挑选程序员时，用的是掷骰子的办法。"那么给他的主管留下的印象，显然就不如前一种提法好。

但是现在，我们已经有了针对程序员的人格模板（Perry & Cannon，1966）——只要人事主管确实能够使用这种人格模板，效果应该会比简单地掷骰子更好些。但是，其中还是存在种种值得怀疑的地方。比方说，我们正在讨论的是哪种类型的程序员？或者说，针对包括系统程序员、应用程序员、产品测试程序员以及系统维护程序员等等在内的各种不同类型程序员，我们是否应该在人格模板上对其加以区分呢？

纵然我们在定义程序员人格模板时，可以细致到对不同的工作类型进行区分，难道这些模板所反映的确实就是我们所希望的吗？即使我们知道自己所要的是什么，这些人格模板也毕竟只是通过对已经加入该行当的那些人做的测试而建立起来的，而并不是我们期望加入该行当的那些人。对于那些历史久远、已经定型而且稳定的职业，还姑且可以假定说，正在从事这些职业的人中，绝大部分都没有入错行——尽管这些人很可能就是在斯特朗职业兴趣调查表的"引导"下，才最终做出了这些选择，并使其自我实现的梦想得以满足。但是可以肯定的是，就目前我们对程序开发的了解而言，仍然十分有限。相反地，我们有充分的理由相信：即使是对于一般水平的被雇用的程序员，也至少不是所有的雇主都会感到满意。既然如此，为什么还要对照从这批人身上得到的人格模板，在应聘者中按图索骥呢？

关于这个问题，还有一点值得讨论——这就是所谓的"作弊"问题。就其本质而言，包括人格测验在内的所有心理测验都建立在这样一个假设之上：设计测验的心理学家要比接受测验的人更加聪明。有些人会热衷于从事人格测验这份职业，如果能够列出这些人的一份名单，我们就会发现，通常他们的确会努力使自己比别人更为聪明。或许，除此之外他们别无选择；要不然，他们地位之特殊又将如何得以体现呢？但是，

一个自以为聪明的人，就一定会确实比别人更聪明吗？

从某种意义上讲，人格测验其实就是一种智力测验——这是一场在被测与测验方法的设计者之间进行的一场智力竞赛。比如设想一下，如果为了应聘一个程序开发的工作岗位，某个人需要参加一个测验。在测验中，他需要就下列陈述表示同意或者反对："每当面对一个问题绞尽脑汁、苦思冥想的时候，我常常会头疼，并因此无法继续工作。"要是真有人傻得可爱到回答"同意"的地步，那么他可能确实不能胜任这份程序员的工作。正如我们所知的，前来应聘程序员工作的申请者，似乎都是一群更为聪明的人；所以，如果他们需要参加这类的测验，我们可以想像得出他们会如何地"作弊"。但是，我们也不必担心。因为，如果他们果真能够成功地作弊，也许他们正好具备了我们所需要的一些人格特征，而且这些特征往往非常重要。这些特征包括感知测验动向的适应能力、面对陌生人的考察同时又不得反抗时对压力的承受能力、行事果敢的能力以及把测验的挑战当成一件乐事的幽默感，等等。

当然，心理测验的执行者们不会承认存在这种作弊行为的可能；如果有人指出这个问题很重要，或这种现象时常发生，他们只会嗤之以鼻、一笑了之。但是在程序员当中，这种作弊的行为的确会出现。曾经有一家大型公司，在聘用程序员时，专门聘请了另外一家心理咨询公司来设计相应的人格模板。依照惯例，要根据一批当时已经在该公司工作的程序员为原型，将这个模板设计出来。不同寻常的是，这批作为原型的人，都是在公司中被公认为"最出色的"程序员（当然，也许他们自己也根本不清楚，为什么要奢望每一名参与程序开发的员工都是"最出色的"）。

为了建立人格测试的模板，被试程序员都是按照一定的标准挑选出来的，其入选的条件是：他们必须曾经被挑选出来，至少参加过一次公司级别的程序员大会。本来为测验预留了好几个小时，但是当进行测验的心理学家来到测验现场时，他才发现参加测试的人有 60 人之多，而他预计的人数是 30。虽然他没有带如此多的测验表格，但是幸亏测验用的人格问题列表分为两部分，所以他先让其中的一半人先回答 A 部分，而剩下的一半人回答 B 部分。在前一小节完成后，两部分人要交换问卷。接下来心理学家向被试做了一些例行的交待，但最后他却提了一个冒失的问题："还有

什么问题吗?”这时，居然有人举起了手——由于测验都已经进行了一半，所以这位心理学家对此惊诧不已。

心理学家努力使自己重新平静下来，让提问者讲话。这位程序员的问题直截了当、毫不掩饰：“你是否认为我们在应付这一半测验时，应该使用与前一半相同的人格?”

面对突如其来的刁难，心理学家的适应能力似乎还不如一位普通的程序员。他的脸涨红了，吭哧了半天才勉强重新开口说话，但这次的语调却犹如一个受委屈的小孩子：“我们希望你**诚实地**回答问题，尽你所能地诚实。”

“你把我们当成什么啦？傻瓜吗?”另一名程序员回答道。顿时，屋子里响起了一阵诡秘的笑声。

程序员的人格测验

通过上一节的讨论，我们已经对人格测试可能出现的问题有了充分的警觉；现在，我们就可以来看看一些针对程序员的人格测试结果。针对通过斯特朗职业兴趣调查表以及其他测试工具获得的结果，David B. Mayer 在 1968 年给出了一份很有用的调查报告。如果你对斯特朗职业兴趣调查表兴趣确实很浓厚，而且并不满足于我们这里的简单介绍，那么你可以从这份报告中得到进一步的启示。Mayer 首先正确地提醒大家：这种工具**并未宣称，它能够对工作绩效做出预测**。实际上，这种工具的作用，仅仅是抽取出关于被试程序员之兴趣点的信息；而至于其兴趣与其适宜的工作之间的关系，则有待于使用者来确定。

Mayer 指出，斯特朗职业兴趣调查表的确能够显示出，在被试程序员与一般人群之间存在某些差别；同时，针对公众的兴趣，他也给出了几个精选的实例。例如，与一般的人群相比，更多的程序员会“喜欢保守稳重的人”，同时很少有程序员会“喜欢乐于变革的人”。通过这些令人赏心悦目的实例，Mayer 成功地激发起了读者的兴趣。但是他突然话锋一转，进而问道：“难道孤立地根据斯特朗职业兴趣调查表中的各个项目，我们就可以说出其中某些部分所指示的是对应于成功程序员的一般性兴趣模式

吗？答案是‘否’。”

对于那些孤立地看待各个项目的做法，为什么 Mayer 会如此坚决地予以排斥呢？第一个原因就在于，为了使得这种工具有实际意义，必须在事后对由 400 个问答回答所构成的模式进行分析。而为了完成这种分析，我们必须使用计分键。而为了得出合理的计分键，我们必须对大量的样本进行统计（比如，由 Perry 在 1966 年获得的结果）。Perry 针对程序员的计分键，并不能适用于斯特朗职业兴趣调查表的所有用户。但是至少根据 1968 年对一些单位进行的一次调查的结果，其中没有任何单位在使用斯特朗职业兴趣调查表；但是在那个时候，这是惟一的一种能为程序员进行计分的人格测验方法。

在 Mayer 进行调查的当时，还有一些针对人格和兴趣领域的其他测验方法，只不过它们极少被采用。这些方法包括：瑟斯顿个性测验表（Thurstone Temperament Schedule）[①]，行为向量分析（Activity Vector Analysis）以及库德偏好测验（Kuder Preference Test）[②] 等等，在当时的 282 家单位中，只有大约 10～15 家在使用这些方法。从我们在程序开发中保留这些测验方法的角度来看，许多公司的决策非常高明，它们似乎都在等待进一步结论的出现，而没有在仓促间就陷入对其程序员进行人格测试的泥潭。

即使有了对兴趣及人格进行测验的正确而有效的办法，人事主管还是会遇到其他的一些复杂问题，比如，针对某种目标应该采用哪些测验？应该按照何种次序进行这些测验？再如，一旦获得结果之后，应该如何进行处理？在这些测验中，无论是管理还是计分都需要经费支持，因此为了说明它们的确有用，主张这些测验的那些人就必须去刻意证明，这类测验要优于那些简单易行、更加直观的测验。对于那些可以通过简单的面试就获得的信息，Mayer 称之为 X-要素[③]，而且他猜测它们与智力水平有关。正如我们已经讨论过的，根据我自己进行面试的经验，我倾向于认为 X-要素

① 由美国心理学家 L. L. Thurstone 于 1949 年提出。该测验可用来测验各人日常生活的好恶、偏爱及习惯。答案因个人性格的不同而异，没有对错之分。作答方式为要求被试在“是”、“?”或“否”下打“√”，然后将被试的测验结果与常模进行比较。——译者注

② 又称 Kuder Preference Record，由美国心理学家 G. F. Kuder 于 1939 年提出。——译者注

③ 意为“未知因素”。——译者注

更加侧重于人格因素。为了证明自己存在的价值，任何一种心理测验工具都必须比 X-要素更好——而且这种优势必须是确凿的、可以证明的。

为了能够借助心理测试来选拔人才，人们曾经做过许多努力。关于其中的一项重大的努力，有一个经典的故事。虽然这个故事或许属于野史，但是沿着上述思路，还是值得做一引述。在第二次世界大战期间，美国军队所遇到的一个问题就是，他们必须在情况迥异的不同前线作战。美国人很快就发现，有些士兵在热带地区总是斗志低迷，而另一些人在寒冷的气候下就是提不起精神。为了给对应于不同军事行动的各个战区挑选最适宜的士兵，美国人进行过一项大规模的研究。为了得出气候与士兵在战场上的表现之间的关系，用成百上千的问题对他们进行了测验。等到这项研究有所“成果”时，战争正好结束，所以其成果没有来得及得到实际应用。但是根据这项研究结果，所有的问题中只有一项与气候对战场行为的影响有重要关系。而这个问题究竟是什么呢？非常简单：“你更喜欢哪种天气？炎热的？还是寒冷的？”有没有人曾经想过，面对一位前来应聘者，我们是不是应该首先问问他们是否喜欢程序开发的工作呢？

小结

正是由于程序开发任务与生俱来的复杂性，所以程序员的人格——其个性以及特征——对其工作成功与否的影响之大，远远超过人们通常的估计。然而，在挑选将来的优秀程序员方面，人格测验并未获得成功。导致其失败的原因，一部分可以归结为测验本身的局限，另一部分可以归结为我们对程序开发过程本身的理解不足，还有一部分则可以归结为我们对“哪种人格因素在程序开发过程的哪个部分中会起到什么作用”这一问题理解不够。

尽管如此，还是有证据显示，关键的人格因素可以被分离出来，并且与特定的程序开发任务之间存在联系。缺少其中某个因素，程序员都可能无法顺利完成某类任务——至少从这个方面讲，二者之间确实应该存在联系。因此，无论是心理学研究者、主管或经理，还是程序员本人，

只要对人格问题有所重视，都将会从本质上有助于程序员工作绩效的提高。

思考题

请主管回答：

1. 你是否见到过手下的哪位程序员，其人格在突然间发生了变化？你当时是如何处理的？如果今天再遇到这种事情，你的处理办法将有何不同？

2. 在选用程序员时，你所期望的人格特征是什么？在对程序员进行评估时，你欣赏哪些人格特征？这些人格特征是否相同？如果不同，为什么不同？

3. 在选用程序员时，你所在的单位是否采用人格测验的方法？如果是，使用这些测验（包括你自己在内）的人，是否在解读测验结果方面受过培训？你是否了解为什么这些测验能够在选用程序员时有效？你是否曾经试图向自己所在单位证明其有效性，还是仅仅自己了解就足够了？

请程序员回答：

1. 作为一名程序员，你认为人格因素中最有用的是哪一方面？你是否认为，你的雇主在聘用你担任现在的工作时，也认为这个方面是最重要的？

2. 作为一名程序员，你认为人格因素中最有害的是哪一方面？在你被聘用时，这个方面是否被考虑过？你正在采取哪些措施，以消除这方面因素对你的工作绩效产生的负面影响？

3. 请你的同事们为你回答一下第 1 和第 2 个问题。他们的回答与你的回答是否一样？从同事们的意见中，你是否能够了解到自己的什么问题？你决定如何加以解决？

参考资料

- Mayer, David B., and A. W. Stalnaker, Use of Psychological Tests in the Selection of Computer Personnel, *SHARE/GUIDE Presentation*, October, 1968.

在由 ACM 人事研究专用业务组（SIG/CPR）资助的第五届 Annual Conference on Computer Personnel Research 上有一个教学演讲，这份综述报告就是根据这次演讲改编而成。这篇论文对该业务组自 1962 年成立以来的工作做了一个很好的总结，并且提供了一份很好的参考书目清单。

- Perry, D. K., and W. M. Cannon, A Vocational Interest Scale for Computer Programmers-Final Report, *Proceedings of the Fourth Annual Computer Personnel Research Conference*, Association for Computing Machinery, New York, 1966, pp. 61－82.

- Cronbach, L. J., *Essentials of Psychological Testing*, 3rd ed., New York, Harper, 1970.

这是关于测验的一份严谨的总结。

- Wernick, Robert, *They've Got Your Number*, New York, Norton, 1956.

这是一份关于测验的不严谨的总结，而且正因为它不严谨，所以实际上它非常严谨。

- Cronbach, Lee J., and Goldine C. Gleser, *Psychological Tests and Personnel Decisions*, 2nd ed., Urbana, University of Illinois Press, 1965.

纵然心理测验应用得法、分析正确，主管还是会在做人事决策时遇到问题，因为决策的结果并非由测验的结果惟一确定。或者换而言之，主管必须确定应该采用哪些测验，以及到底是否应该测验。任何决策都不能轻易做出，也没有万能的答案，但是这本书在其中一些问题的讨论上显得过于深奥，甚至常常数学味道太浓。

- Hall, C. S., and G. Lindzey, *Theories of Personality*, 2nd ed., New York, Wiley, 1970.

关于人格的理论真是不少，当然，其中每一理论都只包含一点儿真理，同时也为程序员及其主管提供了一点儿启示。Hall 和 Lindzey 对 12 种最为主流的人格理论进行了公正的

阐释，同时也进行了批判的分析。对那些心理学的门外汉而言，这本书是引导他们进入心理学领域很好的启蒙教材。

□ White, R. H., *Lives in Progress: A study of the Natural Growth of Personality*, 2nd ed., New York, Holt, Rinehart and Winston, 1966.

在人格培养方面的研究工作，很多都集中于正在成长的儿童方面。除非我们需要招聘程序员童工，否则这些研究结果对我们没有多少用处。White 的与众不同之处在于，他的兴趣集中在成年人的人格发展方面。他所列举的示例人物最早从其大学毕业时起，而由于这正是许多程序员首次择业的时期，所以程序开发主管们应该会对这本书产生不同寻常的兴趣。

□ White, R. W., *The Abnormal Personality*, 3rd ed., New York, Ronald, 1964.

对于那些人格令人难以捉摸的人，有些程序开发主管会马上给他们贴上“疯子”的标签。为了让这部分主管在此之前能够三思而行，White 特意编写了这本切合实际的教材。书中列举了若干人的实例，这些人的人格已经超出了“正常”的范围。这类书籍的主张与连篇累牍的管理“心理学”丛书截然相反，因此对废止所谓的“快速心理疗法运动”很有帮助。

第 8 章　人格因素

评　　注

要是我今天从头开始写作《程序开发心理学》的话，这一章将是内容需要改变最多的，因为在过去 25 年之中，这正是我通过学习收获最多的一个方面。首先是因为，Isabel Briggs Myers 于 1980 年发表了她的麦布二氏性格取向指数测验（Myers-Briggs Type Indicator，MBTI）理论[①]。在读到这本书之后，我放弃了所有其他的人格测验理论，因为将它们应用于程序开发之中实在过于牵强附会。接下来，在 1984 年 David Keirsey 和 Marilyn Bates 共同推出了他们关于气质的相关理论[②]。与 MBTI 一样，这一理论所讨论的也是“正常”人格之间的差异。从那时起，我与其他人就开始将这种方法应用于对软件商业环境中的人格问题的研究[③]。

在此期间，我还师从 Virginia Satir，学习了有关“非协调”人格态度——也就是当承受的压力超过“正常”范围后，人们应对压力的方式——的知识。MBTI 和 Satir 关于应对策略的理论，给我在人类行为方面的思维方式、培训以及咨询工作方式带来了一次革命。

① Isabel Briggs Myers, *Gifts Differing* (Palo Alto, Calif.: Consulting Psychologists Press, 1980). 出版社联系方式：577 College Avenue, Palo Alto, CA 94306.

一种心理测验方式，主要从四个方面对人的个性类型进行分析，并分为 16 种个性类型。——译者注

② David Keirsey and Marilyn Bates, *Please Understand Me: Character and Temperament Types, 4th ed.* (Del Mar, Calif.: Prometheus Nemesis Book Co., 1984).

③ 请特别参见：*Quality Software Management*, Vol. 3，其中详细讨论了 Satir 的应对态度理论。

在本书的第1版中，我曾经预言：

> “因此，无论是心理学研究者、主管或经理，还是程序员本人，只要对人格问题有所重视，都将会从本质上有助于程序员工作绩效的提高。”（p. 217）

尽管当时的我对人格几乎没有任何了解，但是这句话后来却被证明是我所做过的最成功的预言之一。

第9章 智力水平以及问题求解能力

正如我们所看到的，如果你与程序员一起工作，那么你就应该意识到自己的周围是一批智力超过常人的人。尽管还没有见到过这方面的正式研究成果，我们还是可以不过分地估计：程序员平均的智商甚至要超过大学研究生；而且大体上讲，一个程序员越是成功，其平均智商也会越高。当然，智力水平并不能代替所有的方面——很早以前我们就摒弃了这种谬论。然而大多数程序员的智力确实要高于一般人的水平，因此在对程序开发这一过程进行研究时，无论“智力”一词的含义具体如何，只要我们能够着眼于智力水平对程序开发工作的影响，那么就可以更好地理解该过程。

心理集合

对于某些类型的错误定位工作来说，心理集合（psychological set）[①]已被证明是一种障碍。一系列的实验结果已经证实，人的眼睛总是倾向于看到自己所期待看到的东西。例如有一个实验是这样的：在一组单词中放入一个不是“单词”的单词（比如“dack”），然后让被测快速地浏览一遍这些单词。心理集合的第一种影响就表现在，被测会把这个不是单词的字符串当成单词。其原因在于，在他快速浏览的时候，他有一种把这种字符

① 人们在思考问题时，常会受制于自己头脑中既有的、僵固的想法或方法，这种情况称为心理集合。这种心态会严重妨碍我们去寻求其他的、新的解决问题之道。——译者注

串看成单词的倾向（或者称为心理集合）。而如果被试在事先被告知这些单词都是有关“兽类或鸟类”的，他就极可能会把“dack”看成“duck”（鸭子）；而要是反过来，如果他事先被告知这些单词是有关“交通工具”的，那么他常常会把“dack”当成“dock”（码头）。——这是心理集合的第二种影响。

如果曾经做过校对工作，那么你应该已经意识到了这种心理集合的现象；而要是你有在程序中查找打孔错误之处的经历，那么你对这种现象就不仅仅是意识到了——也许，这件事在你心中已经留下了难以磨灭的烙印。在诸如校对等工作中，克服心理集合带来的影响到底有多难，这种困难本身就很难度量。因为，为此我们需要一些文本样例：其中有些文本必须十分完美，以作为标准；而另一些文本却需要有足够的难度，以模拟实际的情况——但是，这些文本很难得到。但是在计算机程序开发过程中，情况则不然。因为，计算机可以为校对工作的有效进行，提供一个自动的测试平台。

计算机可以为程序本身的这类错误提供测试手段，而且这类手段在很多方面显得既细致也强大。就其简单的方面而言，包括对无法识别的语法错误、关键词拼写错误以及所有形式非法的常量进行检查。基于符号表、交叉索引表以及数据流分析等的交叉核对，就要更深层一些，但是就其本质而言，它们仍然是静态的。对未经初始化的变量、流跟踪以及子程序调用跟踪的动态检查，可以清除这类排版错误，而不必在其他层次上逐字逐句地筛选。但是，还没有任何自动的系统可以确保找出所有的这类错误。如果我们在编写程序，甚至是设计语言时能够对心理问题有所重视，那么相信情况将可以得到一定的改善。

与“趋向”这个概念相关的是“距离”的概念。不管读者的心理集合有何差异，并非所有的上述误读现象都是相似的，所以与“dack”相比，“daxk”更不容易被误认为“duck”——因为，这时读者需要变换两个字母，而不是一个。根据信息理论，两条信息——无论是字符串还是比特序列——之间的“距离”被定义为二者的比特位表示之间的不同比特位的数目。这种度量方法的重要之处在于，它可以用来衡量，为了从一条信息变换到另一条（比如，当信道中存在噪声的时候），需要更换的比特位数。

程序开发过程中使用的语言符号，正如自然语言中的单词；对于这种符号来说，上述关于“距离”的定义过于简单，以至于在确定一个符号不被误判为另一个符号的可能性时，这种简单的估计只能作为第一级的近似。比如，已经有心理测试表明，在区分两个单词时，首、尾字母的作用更为重要。因此，虽然就其与单词“duck”相差的字母数目而言，“gucr”与“daxk”都是相差两个字母，但是“gucr”更不容易被误判为“duck”。此外，每一对符号之间的距离，和其他符号对之间的距离不能视为等同。虽然这种精确的关系将取决于具体所使用的打字稿，但是与字母对“x”与“o”相比，字母对“x”与“k”更加容易被搞混。

每一名新入门的程序员所上的第一课就是，为了方便其他人为自己输入的程序，要学会如何在手写体中仔细地区分“数字零 0”与“字母 O”。但不幸的是，大多数的程序员在此后就停滞不前了；而如果他们继续培养这种良好的习惯，就可以在程序开发过程中，为克服来自心理集合的风险提供便利。例如，无论书写者如何仔细地把符号“ST0P”中的数字零特别地标明出来，它还是会在各种地方被误认为字母 O。符号“ST0P”与“STOP”之间的心理距离非常近，这不仅是因为数字零与字母 O 外形酷似，以及它们所处的那个位置是二者惟一不同的字母，而且也因为在该符号中**所蕴含的**心理集合——所有这些加在一起，很容易诱导我们把它理解成一个英语单词。但是，它们之间的距离是如此之近，以至于那些习惯性地做这种选择的程序员，接下来总是会面临某种灾难。

毋庸置疑地，正是由于有所谓的自动方法来检测这些错误，而且这些方法也的确取得了广泛的成功，所以将许多程序员引入了歧途，以至于他们在选定符号时总是粗心随意。但是，在一些情况下，无论编译器还是解释器有多么敏锐，都有可能遗漏这类错误；而且，这种情况总是会出现。有这样一个例子：某位程序员在同一段代码中同时使用了“SYSTSTS”和“SYSSTSTS”两个符号。但是最后才发现，其中一个已经被替换成了另一个——而这时，模拟程序已经被错误地运行了几百个小时，所得到的数据已经被收录到一本书中并已经出版；而且，还根据这些结果错误地设计了好几个系统。其实，要是他此前能够遵循在不同符号之间保持一定的最小间距的原则，并且能够确保其差别至少有一处出现在符号的首或者

尾，那么所有这些后果本来都是可以避免的。

在阅读程序时，助记符号尤其容易引起读者的麻痹。助记符号之所以会使我们产生误会，有以下几个原因：

1. 它们容易使程序看起来显得是“有意义的”。但是，它们只不过是满足了人们通常的某种心理集合，这种心理集合总是会将无意义的东西看成有意义的。

2. 人们会倾向于相信助记符号的名字，而不是其名字所代表的真实含义——助记符号正是利用了我们的这种心理集合。有谁会相信，符号“FIVE”代表的居然是数值4？但是这种事情的确曾经发生过。当时，有一位程序员需要修改程序，但是又没有足够的时间更换所有对变量“FIVE”的引用。但是，他还是有足够的时间来再次运行该程序——虽然是在花费了大量时间才确定问题所在之后。

3. 使用助记符号，也很可能给出一些毫无优化“距离”可言的符号式样。比如，英语单词并非是由字母随机堆砌而成的。有些诸如“辅音—元音—辅音”或者“辅音—元音—元音—辅音”的音节式样会时常出现。更为糟糕的是，英语中还有所谓的同形异义词，比如“LEAD”（领先）和“LEAD”（铅）。来自不同程序的同形异义词，可能会汇合到同一个程序中。

4. 由于我们进行缩写的方式变得日益规律，所以最佳的距离被进一步缩短了。这可能会造成诸如符号“PEND”的歧义：它有可能代表一个正处于挂起状态的（Pending）记录，也有可能是“P单元的末尾（End）”。

在结束针对心理集合所导致错误的讨论之前，我们不能不再谈谈注释的问题。之所以要在程序中添加注释，是为了让读者能够正确地理解相应的指令或者语句。只有在注释所说明的代码本身是正确的时候，这种方式的注释才会真正有用；然而，一旦代码本身有误，由注释所引起的心理集合，只能使得代码中的错误更难被发现。

由Okimoto在1970年完成的一个实验，可以很好地说明在你阅读的程序中存在错误时，其注释会给你对代码的理解造成什么影响。在这个实验中，针对同一段有错误的代码，给出了几个不同的版本：其中之一有正确的注释，另一个版本有一到两个错误的注释，而最后一个版本则没有任

何注释。至少对某些类型的代码，在没有任何注释的情况下，程序的功能反而可以更容易被人理解，而且理解的程度更为可靠，理解的速度也更快。如果注释书写合理，那么在提供正确注释时，要比提供的注释有错误或者有歧义时，程序中的错误会减少得更多。尽管如此，在查找程序中的错误时，许多经验丰富的程序员还是有一种把所有注释都遮盖起来的习惯。这种习惯的好处在于，可以尽量地减少心理集合的作用。虽然这种作用对理解正确的代码很有帮助，但是如果你正在调试程序，那么它就会把这项本来就令人难以忍受的工作搞得更加让人头疼。

问题求解的一些维度

在心理学中，“趋向”通常被看作为与“直觉”有关的研究的一部分内容，而不是“智力”的一部分。但是从前一节的讨论可以清楚地看出，对于那些我们肯定会称之为“问题求解”的行为，趋向现象也可能会对其产生影响。当然，在考虑关于问题求解的问题之前，首先需要考虑的应该是对问题**回避**的问题。正如我们所见到的，存在很多现成的技术，使得程序员可以在某些情况下完全避免心理集合所引起的问题。请从理论上考虑这样的一个断言：和那些被问题缠住的程序员相比，无论他们最终是否确实能“解决”问题，那些能够完全回避这一问题的程序员要更“聪明”。

但是，在理想中关于智力的构想，很难与我们对于实际情况的理解完全一致。在缺乏客观测量标准的前提下，我们常常会通过程序员在一个程序上的出力程度，来判断该程序的难易程度。一旦采用了这类测量方法，我们就很容易陷入这样一个悖论：最差劲的程序员恰恰就是最优秀的——因为他们工作得最卖力。有这样一个例子可以证明这一点：在某个新的计算中心，为了使某个小型程序可以运行，一位程序员每周 7 天、每天 14 个小时地持续工作了八个星期。鉴于他的努力程度，其所在的公司给他颁发了一个“额外加班工作”的奖励，并且还将他提拔到管理层。此后不久，另一位程序员被指定为该程序扩充一些功能。但是这位新程序员却发现，原来的程序简直就是一堆混乱不堪的垃圾，与其对它进行修修补补，

还不如重头写过更加省事。

重新编码和调试工作只花了一周就完成了，而且该程序员只占用了正常的工作时间。即使说重新写一个程序会比第一次编写该程序要简单，但是前后这两次开发在时间代价上的差别还是太大了。而且在运行速度上，新的程序要比原来的程序快 8 倍；在存储方面，新程序节省了一半的空间；而就代码本身的大小而言，新程序的行数也只有原来的一半。很显然，第一位程序员之所以受到嘉奖，仅仅是因为他的小题大作。而由于管理层的那次奖励不公，严重地挫伤了整个程序开发小组的士气。

因此，最极致的回避问题的行为是很明智的行为；但是如果是为了吸引其缺乏训练的主管的注意力，那么一个人的这种行为就显得不甚明智了。那些**没有**落到我们头上的麻烦总是很难引起我们的重视——这就好比在进行问题求解时所做的出色工作也总是难以引起人们的重视。一旦问题的答案已经揭晓，人们就会很容易把在解决该问题之前自己所面临的困惑扔到九霄云外。首先，一个问题之所以会很难，最常见的困难之一就在于我们可能忽视了一些因素。一旦我们自己发现或者是被告知说某个因素很重要，那么求出最后的答案就变得轻而易举了。如果我们转而向其他人提出该问题，我们通常都会同时提供关于这些因素的提示，而这种提示一下就会让他解决问题的 90%。这时，他就会觉得很奇怪：既然如此，为什么你们当时会感到那么困难？而且过不了多久，我们也会开始怀疑自己的智力水平。

在导致我们迷路的各种事先人为假定中，对某方面因素的忽视只是其中的一个特殊例子。也许根本还没有对某个因素做认真的考察，我们就可能会匆匆假设该因素并不重要。但反过来，如果我们事先就假设某个因素**是重要的**，而实际上该因素对我们手头的工作没有多大的影响，那么也类似地有可能使我们迷路。如果一名程序员在调试别人的程序时花费了很多的时间，那么他很快就会明白：在解决问题之前，不要去听任何的解释。——因为，这些解释会很容易将倾听者引入解释者的老路上去；而解释者原来在这条路上之所以没有成功，是因为他曾经做过一些错误的假设；但是通过倾听，新来的程序员也会接受这些错误的假设，于是必然会重蹈覆辙。

当然，心理集合就是设定假设的另一种形式。尽管这种假设会隐藏得更深，它们对问题求解的作用却是相同的。因此，难道我们不能说，问题求解的第一条法则就是“千万不要有任何假设”？

我们的确可以这么说，但是，这恰恰是错误的。实际上，只要我们希望在问题求解时获得成功，我们就**必须**进行一些假设。如果我们将自己遇到的每个问题都看作全新的问题，那么我们在问题求解方面的绩效就不可能得到提高。比如说我们的心理集合，虽然它可能会误导我们把印刷错误之处当成是正确的，但是在大多数情况下，这恰恰是我们最为宝贵的优点。只有在进行校对之类的工作时，这类心理集合才会给工作造成困难，然而实际上很少会有人需要花时间去进行这类工作。因此，真正高明的做法并不是把所有假设统统抛弃掉，而是要在必要的时候，能够充分灵活地利用各种假设。换而言之，所谓高明的人，并非有什么万能的解题模式可以适用于所有问题，而是拥有一系列的“解题模式”，但是他们对其中的任何一个模式都没有偏爱，所以不至于出现因为对某种模式割舍不下，而忽视了最适用于当前问题的那种解题模式。

在我们结束关于问题求解过程的适应性问题的讨论，并进而就一些被成功的问题求解者有选择地采用过的“解题模式”做详细介绍之前，我们还必须努力摒弃关于智力的另一种谬论。无论如何终极地对它做定义，智力只是一个从统计上讲才有意义的概念。我们永远也不可能只凭借其在某个特定问题上的表现，来断定一个人的智力水平。实际上，借助很多的非智力因素，同样可以得出“正确的”解答；同时，一些智力因素也可能会得出“错误的”解答。实际上，有些程序员对他们的成功所做的解释，让我不禁想起这样一个故事：一个乡下的白痴居然赢得了一次月度抽奖活动。当被要求解释一下他如何确定中奖号码时，他回答说：“是这样，我的幸运数字是7，而这是今年的第七次抽奖，于是我把7和7相乘，就得到了中奖的号码——63。”

但是，当有人试图向他解释7乘以7应该等于49时，他只是轻蔑地回答道：“噢，你有点妒忌吧？”——很不幸，他当然又一次猜中了。①

① 这一节 Weinberg 对问题求解的问题进行了论述。有关这一方面的更多、更精彩的论述，请参见其后期的著作：《你的灯亮着吗？——如何找到问题的真正所在》。——编注

程序开发的智力因素

根据上面的论述，适应能力是所有类型的智力行为都必须的。有些行为虽然是属于精神方面的，但是其内容不过是简单地执行一组固定的规则，所以严格地讲他们还不属于智力行为。实际上，这种类型的工作与其让人来做，还不如让计算机来完成。然而这并不是说，简单地执行一组固定的规则，不是智力行为的一个**重要的部分**。恰恰相反，如果一位程序员即使是完成把两个数相加这样简单的计算，也要费九牛二虎的力气，那么在设计最佳问题求解方案的角逐中，这位程序员肯定会掣肘于其差劲的算术能力。当然，如果该程序员能够通过建立一系列捷径来回避其在算术计算方面的欠缺，那么他遇到的这种阻碍，就可以转化成为长处。

在很大的程度上，如果只是因为某些人能够比其他人更好地做某些事情，那么应该说每一种问题求解技术都是因不同问题而异的。任何聪明的人都往往会在寻找解决问题的方法时，充分运用其最擅长的方面，同时尽力去回避自己的弱点。关于这种特点，我们的记忆力就是一个对应的例子。

毫无疑问，对于一名程序员来说，记忆力是其智力中最重要的方面之一——如果他确实能够驾驭这种能力的话。记忆力可以在很多方面帮助程序员，而不仅仅是在他没有把所有的资料一股脑儿放在眼前时，使他仍然能够继续工作。有这样一则奇闻轶事，讲的是一位程序员如何躺在床上就解决了一个问题：

> 我昨天早上在计算中心遇到另一位程序员，他向我提了一个问题。据他讲，从去年4月开始，这个问题就一直在困扰着他，而现在已经是1月了。这个问题本身倒不是很严重，但是自从其出现的那天起，这个问题就不时地浮现在他心头，但是无论他用什么办法，就是解决不了。问题出在一段PL/I程序上，该程序的输出格式不太对头。这位程序员编写了一个ON ENDPAGE例程，但是它只能在第一页的末尾处、被SIGNAL例程激活时才有效。但是一旦第一页被输出后，剩下的所有页都依次照常输出，而每一页的标题都没有按照程序的要求输出出来。

我检查了其使用的PAGESIZE常量、任务控制卡片——希望看看他是否使用了奇怪的回车控制条件。我也检查了ON-例程的位置，但是很显然，它只被调用过一次。那么，是否会在其内部还有什么开关？也没有。我所发现的惟一一个非同寻常之处就在于，他是通过PUT SKIP例程来打印各页的标题，而我通常会使用PUT PAGE例程。我告诉他，这就是为什么标题没有被正确地打印在各页顶端的原因。但是，在对其他一些假设进行测试之后，我发现这并非是主要的问题。我告诉他如何通过对LINENO标志进行比较，以判断何时应该激活ENDPAGE标志，从而最终实现输出正确。虽然这些已经可以令他非常满意，但是我自己清楚，只有把问题的根源挖掘出来，才可以让我的头脑停下来休息。

我竭力想忘掉这个问题，但是昨天晚上当我上床睡觉后，我的思路却似乎变得清晰了起来，于是我干脆就决定一边躺着，一边继续研究这个问题。我在头脑中把完整的情况——代码及其输出——都重新构造了出来。当我闭上眼睛，静静地躺卧的时候，我能够回忆起自己正在处理的任何程序——即使是这样一个我只看过一眼的程序。我在脑海中不断扫描程序的输出，并尽力设想：什么样的程序会有这样的输出？我把早晨对问题原因所作的所有假设都回味了一遍，经过严密的思考，这些假设都被逐一排除了。因此，我决定从新的方向着手。

在对程序浏览过一遍之后，我觉得可疑的地方是在ON-例程中对PUT SKIP的调用。随着我越来越频繁地浏览这段代码，这个疑点在我的脑海中越发突出，因为我没有这样写过程序。从来没有？但是在这种情况下，也许正是这一点造成了差别。但是，通过另外一种方式来开始新的一页，为什么就会造成页尾标志的丢失呢？还有，每一页的末尾是如何判断的呢？可以根据从每一页顶端算起的行号来确定。但是，又该如何判断每一页的顶端呢？我意识到自己还不知道确切的答案，因此设想了很多种不同的可能。这一次，我意识到自己思考的方向是对的，因此只是一味地逐一检查各种可能，并设想各种情况将在程序中发生的作用。

最后，轮到了“只有PAGE标志才能重新分出一页”（虽然LINE

> 标志也可能重新分页，但在这里没有用到）这一假设，在我对其作用进行了分析之后，我终于找到了问题的症结：在ON-例程中的对PUT SKIP例程调用，不会重新分页——于是行号计数没有清除，并且与通常一样再次加一，以对应于下一行；而由于根本没有分出新的一页，所以自然就不可能遇到页尾。带着解决问题后的满足感，我很快就进入了梦乡。今天上午，我对这种情况进行了测试，以验证我的猜想，你可以想像出来，情况正如我所预料的。

要是没有良好的记忆力，恐怕这位程序员永远也不能解决该问题，并且从中学到一些新东西。道理很简单，因为他也许永远也不会再次看到这段程序。反过来，如果缺乏这样的记忆力，他也许根本不可能去尝试这样的思考方法。比如，他很可能会与程序员们通常的做法一样，把精力放在生成一批苛刻的测试样例、然后通过计算机来求解问题之上。为了弥补记忆力的欠缺，他需要拿到一份完整的程序清单——这没有什么值得惭愧的。

通过计算机来求解问题，确实可以让程序员在下班回家之前就获得结果，而且不必由于受到来自程序错误的困扰而难以入眠。哪种方式要更好一些呢？我们确实难以孤立地给出回答。比如说，在机时非常宝贵的场合，后一种程序员就会处于更不利的地位；但是反过来，要是所有人都必须加班加点地工作，甚至连睡眠的时间都很少，更别提独自反省的时间了，那么前一类程序员的才能就难以发挥出来。很自然地，最好的方式莫过于，既同时兼有两方面的能力，而且也知道针对具体问题及其条件，应该采取哪种适宜的策略；此外，我们还需要充分发挥出自己的最大潜能。

不同的工作环境，青睐不同形式的智力行为；类似地，在程序开发过程中的不同阶段，也会给不同的程序员提供一展风采的机会。例如，要是正在对一个程序进行总体设计，那么我们最需要的就是一种根据普遍的原则，确定程序设计的新设想并进而对这些设想进行比较和筛选的能力。比如说在结构的匀称性与功能的普遍性之间，就可能需要进行筛选。前者的着眼点在于实现对困难问题的简单编码；而后者强调的是通过简单的编码得出困难问题的解答。然而无论如何，如果在一开始就缺乏设想，那么这些关键的能力就会没有用武之地。人无完人，金无足赤。总而言之，一个程序员无论是缺乏创造能力，还是缺乏选择能力，都会成为其程序设计工作中的掣肘因素。

但是一旦进入编码阶段，其他一些能力的地位就反过来显得突出了。在这种时候，思路开阔而笼统的那些人将风光不再，而那些精于细枝末节的人则会如鱼得水。同样地，等到测试阶段，另一天赋类型的程序员——尤其是那些对系统的整体性（所谓的**格式塔完形心理**[①]）独具慧眼的人——又将走上前台，成为主角。请看下面这则故事：

我当时正在一边吃早餐，一边读着斯彭德·斯蒂芬·哈罗德（Stephen Spender）[②] 的一篇名为《诗歌旨归》（The Making of a Poem）的大作。当我读到第 120 页时，正好是结束了一节，于是我暂时把书放下，给我的粥里加一些糖。然后我重新翻开书，开始阅读下一节，其名曰“记忆”。但是，我很快发现，其第一句话有点问题。这句话是这样的：

“If the art of concentrating in a particular way ……”

我也差不多同时地发现，错误就出在“particular”这个单词上，那里有一个排版错误。但是让我更加迷惑的是，我感觉这个排版错误是把一对字母弄颠倒了，而且我也觉得——尽管这种感觉不那么强烈，但是我确实有这种感觉——哪里还漏掉了一个字母。我核对了一下“particular”这个单词——顺便提一下，我常常把这个词误写为“particluar”——首先是交换一对字母，但是无论如何交换，也找不到这个单词；然后再补上一个字母再试试，虽然这回花的时间更长。但是无论我怎样地努力，都是无功而返。我也确实可以估算出这段时间的长短，因为在此期间，我一共喝了五到六勺粥；通常，每喝这么多的粥，我就需要喝上一口水。但是，当我的手伸向水杯时，我被搞得更糊涂了。

杯子竟然是空的！我于是只好再次放下书本，到厨房去再接一些水。一边往回走，一边喝着水，然后重新捡起书，继续读下去。这一次，我把整段文章都读了下来，再也没有遇到什么新的问题。但是，

① Gestalt，格式塔学说中的完形心理。——译者注

② Spender Stephen Harold，（1909— ），英国作家、诗人。他的诗反映了他个人对于社会及政治不平等的反感情绪；其作品包括诗集《静止的中心》（The Still Center，1935）以及《慷慨的日子》（Generous Days，1971）。——译者注

就在我准备开始下一段时，我的目光又落在下面这一行上面：

"All poets have this highly devolped sensitive apparatus ……"

其中的单词"devolped"也排错了，而这个单词与前面那一段的首句中的"particular"的位置一样。我的感觉没有问题，但是我的注意力"集中"的位置却不对。

虽然这种错误发生在排版中，但是发现并定位这种错误的过程，与程序中的许多错误被发现的过程极其相似。首先，确实有所谓的**"格式塔完形心理"**(gestalt)，即在未经任何定位之前，就认定某些东西的位置有误的一种整体的感觉。接下来的，是一种从难以改变的环境中摆脱出来的能力——这种能力可以改变人们的观察点，甚至会为此而使用一些"盘外招"，比如，暂时离开一下，去取一杯水等等。但是，人们笼统一般的认识终归要发展到对特定问题的认识——这就是此处所谓的"对焦"过程。虽然逐一仔细地对每一行、每一词或者每个字符进行检查，并不能有效地发现错误，但是等到最后，这种细致入微的深入能力必然会变得重要。因此，就程序调试工作而言，需要程序员同时具备这套互为补充的能力。怪不得优秀的调试员总是凤毛麟角！

然而，更为难得的是优秀的文档员。在文档编纂的过程中，在很多方面都可能会出现问题。首先，要是程序本身就写得很蹩脚，再好的文档也不会有妙手回春的神力。而由于程序员往往需要为自己的软件产品编写文档，所以如果要成为一名优秀的文档员，你首先必须是一名出色的程序员——当然，你还必须培养自己运用语言和图表来自我表达的能力。最后，你还必须有足够的耐心，以消除文档中最后的几点含糊不清之处——要知道，最后的5%的努力，总能使你的文档文采倍增。

智力测验

既然智力水平对于一名程序员的成功具有如此重要的决定作用，那么我们应该如何挑选程序员，以保证他们具备胜任工作的能力呢？程序员可能接受的管理性测验形形色色，长期以来主管们一直迷恋于这些测验。虽然如此，多年以来却没有一个人能够证明，在多如牛毛的"程序员智力"

测验中，有哪一项测验的费用（哪怕只是用来印刷的）是值得花的。而如果不力图去揭示出这类广泛存在的问题背后的原因，我们就是失职。

当然，首先程序开发是一项极具多样性的行为，因此根据单一的“分数”，任何测验都无法充分地衡量所需要的智力。的确有一些测试会给出多方面的分数，但是纵然每一项分数各自都是合理的，它们作为一个整体还是很难得到合理的应用。正因为如此，一般的人事主管根本不会对其加以理睬。此外，正是由于能力的多维性，很可能有的人虽然在某些方面的能力差得不可救药，但是在另外一些方面，一旦有机会发挥其专长，他们却又很可能出类拔萃。另一方面，有些人可能在所有的方面得分都很高，但是在实际工作中却表现平平。其原因在于，可能他缺乏某些“次要的”能力（比如与其同事融洽相处的能力）。

不过，因为没有人对某项测验的合理性做过综合的考察，所以这些都是理论上的猜测。与此最接近的一项研究成果显示，如果一名学生在测验中得分很高，那么在随后的程序员培训中，他的成绩也会更好。但是实际的程序开发工作的本质与培训迥然不同，所以等到这个阶段，测验的得分与实际工作的绩效之间没有任何相关性——有时即使有一点点，也是负相关性。

这一令人遗憾的情况并非只在程序开发中才有。正如著名的智商测验一样，智力测验通常只能对一个人在校期间的表现做出成功的预测；但是在其他方面，测验的结果没有任何意义。实际上，正如某位有识之士所指出的，智力测验的得分只说明了被测应对该测验的能力。我们有理由相信，这个论断并非无稽之谈。例如在智商测验中，反应速度是一个重要的因素。在校园环境中，通常也要求这种速度；但是，如果是在办公室，一个小时与70分钟的差别仅仅意味着10分钟——而并不意味着得分为A或者C。迟缓和稳定的人不一定会在跑步比赛中获胜，但是程序开发与赛跑并不是一码事。

智力测验的另一问题就在于，它们只强调短期的——而不是长期的——记忆。如果不去除其所受到的管理限制，那么就别指望它们能起到其他的什么作用。在一项智商测验中，要求被测强记一些毫无意义的单词，或者随意的序列。但是在“真实的生活”中，只有**选择性的**记忆才有作用。这种选择能力，可以使我们忽略不重要的信息，同时在很长的一段

时间内保留住重要的信息。但是校园内的环境却不是这样，于是我们的智商测验与在校的成绩就越来越密不可分——但是这种南辕北辙的做法，只能使我们离自己真正关心的东西越来越远。

最后，确实可以证明，智商分数、程序员的智力分数与某些形式的培训之间具有相关性——这一事实将导出我们的下一论题。在大城市中，总有一些老师承诺可以让孩子的智商提高多少多少分；而许多望子成龙的父母，的确会心甘情愿地花上一大笔钱，来聘请这些老师。同样地，正因为很多公司在招聘时都过于依赖智力测验，所以程序开发培训学校会使用大城市中老师们的方法，提高其毕业生的智力测验成绩，以帮助他们找到工作。因此，纵然有一根魔杖能够为程序员指引未来发展的最佳方向，也无论我们有多大可能会得到这样一根魔杖，我们早晚还是必须丢开魔杖，学会独立工作。

程序员的智力测验

根据上面的讨论，我们很自然地就转到了另一个话题：为了衡量程序开发方面的智力水平，人们专门针对程序员而采用的智力测验。也许此类测验中最重要的，要数所谓的程序员能力倾向测试（Programmer's Aptitude Test，PAT）了。实际上，这种测验并非是一成不变的，而是有数目不详的一系列变种。这些变种的测验都建立在一个最初的测验之上，而最初的这个测验则是由 IBM 公司的纽约科学计算中心（New York Scientific Computing Center）的一组程序员，在 1956 年之前的一个下午设计出来的，并且在那里应用于所有的求职者以及感兴趣的访问者。之所以有那么多的变种测验，一个原因就在于该实验已被广泛使用，而且没有得到保护，以至于没人说得清楚：到底有多少人曾经参加过这种测验（或者更具体地，有哪些人参加了这种测验）[①]。

但是，即使有如此之多的变种，它们之间的差异也不足以保证测验结果的客观性，因为人们总是可以从一次又一次的测验中学到一些东西。在

① 因此，人们只能通过不断对该测验进行调整，以避免因被测多次被测而导致测试客观性的下降。——译者注

涉及IBM以及其他公司中程序员的一系列课程中，我曾经让学生们报告一下他们接受各种变种PAT测验的次数，以及他们的得分。鉴于这种环境是“安全的”——因为这不会与任何人的工作有利害或得失冲突——我们有理由相信，其回答不至于很离谱。总共有27人曾经参加过至少一次PAT测验，其中23人在最后一次测试中获得了A——这些人所供职的公司，都把测验得分为A视为是否雇佣某人的一个重要准则。这些人在首次参加该测验时，有12人得到A，有7人得到B，另外4人得分为C。换句话说，每个人在最后一次测验中的得分，都不会比第一次的得分差。

有这样一个典型的例子。故事的主人公是一位在IBM担任程序员的姑娘。她大学毕业时获得的是数学学位，然后申请到美国无线电公司(Radio Corporation of America，RCA）参加程序开发工作。该公司对她做了PAT测验，但是她的得分是“很低的”B。她因此没有能够获得这份差使，于是她接着去参加IBM的面试。IBM所采用的变种测验与RCA的差别不大，所以她很轻松地获得了A。当被问及“此前是否曾经参加过PAT测验”时，她回答说“没有”，于是她就被雇佣了。

如果具有PAT测验的经验，那么被测的成绩就不能准确反映其真实能力；也许正是由于这个原因，研究人员一直无法把PAT的成绩与实际的工作绩效联系起来。虽然也有过一些关于二者之间相关性的报道（参见Reinstedt，1964），但是我们还是要十分小心，不要被报道的例子中所谓“很显著的”那些结论所迷惑。比如，关于PAT得分与主观评分之间的相关度，在这些实验中最好的结果是0.7。但是，0.7的相关度到底意味着什么呢？

就单一一项研究而言，两个变量之间0.7的相关度意味着$(0.7)^2=0.49$，或者说每个变量的偏差中有49%是由于对方造成的——尽管并没有指明这49%是其中的哪些部分。因此，尽管所谓的相关系数被说成是“解释”，但是这不是全部，还是有一半以上的偏差没有得到解释。因为相关系数总是要大于其平方，所以实际上这方面的因素无形之中被夸大了——这也就是为什么大家都会热衷于使用相关系数的原因。

这种衡量相关程度的方法还有另一种缺陷：虽然其引用的分数表面上看非常明确，但是它实际上取决于主管者心中主观的排名。即使退一步说，这种排名的主观性也至少决定了在衡量程序员工作绩效时的不确定

性。实际上，目前根本没有任何方法或手段，可以用来衡量程序员的工作绩效；无论如何，程序员的工作绩效不可能被归结为单独的一个数字。因此，PAT 得分不可能与任何真正可靠的指标相关联。即使 PAT 得分可能被用来预测程序员的工作绩效，但是由于主管人员本身缺乏培训，他们也看不出什么道道。

在使用这类计算相关度的方法时，我们还要警惕另一个问题。没有人知道，有多少人有多少次曾经试图将 PAT 得分与工作绩效关联起来。但是由于人们期望从实验中发现某种“显著的”关联性，所以虽然更多的人在实验中并没有得到所期望的东西，但是他们往往不会发表这些结果。已经发现在测验得分与在校期间成绩之间确实存在关联性，所以我们了解类似的有关实验（Biamonte，1964；Gotterer，1964）。但是正如你已经看到的，只要数据本身是随机的，而且关联的次数足够多，那么我们最终总能根据自己的意愿，得到任意高的关联性。所以我们可以说，也许要经过上百次的实验之后，才会偶然地得到一次 0.56 的相关度；但是如果不知道总共有多少次试验的结果没有被报道，我们就无法对此类声明做出评估。不过，我们还是可以确定，PAT 的确在成百上千的场合得到了应用。

进一步讲，即使进行过的实验为数不多，在经过上百次的实验后才获得一次 0.56 的相关度，那么这意味着实际的相关度为零。因为实际上，这样一个有水分的 0.56 很容易获得——只要有一个很小（比如说 0.1）的正相关度，那么出现一次 0.56 相关度的概率要远远高于 1%。即使是在这种情况下，0.1 的相关度也不过意味着，1% 的偏差就可以用这种相关性来解释——而仅仅凭借这样一丁点的信息，我们不足以正确地做出人事决定。

即使假设这种相关性是可靠的，根据记录在案的文献我们可以知道，也没有任何比 PAT 更好的手段。不可否认，对于程序开发培训班上的成绩，PAT 的确可以极准确地做出预测。但是在招聘程序员时，这些并非我们真正所希望得到的东西。Mayer 在 1968 年的论述精辟地说明了这个问题：

> 一种非常可能的情况是，如果你在选择人才时使用了所有的测验，你能够（原文如此）挑选出将来极有可能会出色地通过你的培训课程的人。但是有一个问题一直悬而未决：这个人是否真的会热爱程序开发的工作？或者说，这个人是否具备某种能够将他在培训中获得

的成功带到以后的工作中去的积极性？

与其说这类指责的意思是指这些测验目前的应用或许还没有问题，还不如说其真正的意思是：除了纯粹的“智力”因素之外，一个人的工作绩效还会更多地取决于其他因素。虽然我们不会针对这一结论展开争论，但是我们的确认为，有一种可能性被忽略了——具体地讲就是说，这些测验根本就不适宜于考察程序员的智力水平。纵观 PAT 的发展历史，每个人都会感到奇怪：为什么年复一年地，成百上千的公司都会盲目地使用这种测验？只要对该测验本身的结构稍加考察，我们就可以发现到底问题出在哪里，而且我们现在的确发现，其中**问题**颇多。

最初版本的 PAT 测验包括三个部分：几何形体之间的关系规则（这与标准的一套智商测验别无二致）、算术推理以及数列。在几乎所有的 PAT 测验变种中，都保留了前面两部分；而第三部分的数列，则常常被换以字母序列。为了对其中可能的问题有所认识，不妨让我们来考察一下字母或者数字序列的问题。下面是一个典型的这类试题：

1　　4　　7……

被测的任务是补上数列中的下一个数字。不需要很高的智力，就可以说出答案是“10”。但是，这种试题的目的却是似是而非的——它所考察的能力，难道确实是一名程序员所最需要的吗？

我知道有一个例子可以很好地说明，一位优秀的程序员是如何处理这种数列的。在一段 FORTRAN 程序中，包含了这样一条语句：

DO 15 I = 10000, 30000, 10000

这段程序会进入奇怪的循环，虽然循环最终会结束，但是其输出却莫名其妙，没有人能够读懂。当这名程序员来调试这段程序时，他先是沉思了片刻，然后问自己：在什么样的情况下，这条 DO 循环语句不会生成下面的数列：

10000　20000　30000……

突然间，他意识到这些数都是用 15 个比特字宽的寄存器来存放的，于是数列中的下一个数应该是 7232[①]，而整个数列将会是下面的样子：

① $7232 = 40000 \bmod 2^{15}$——译者注

10000　20000　30000　7232　17232　27232……

这恰恰就是程序出错的原因。

换而言之，PAT 测验所测试的，只是针对数字或者字母序列中**常规的**模式的领悟能力。但是对于程序员来说则不然，他们恰恰需要寻找那些**非常规**（甚至是反常）的模式，而在调试阶段，这个特点更为明显。为了考察程序员的智力，类似这样的问题也许会更适合一些：

“给定数列：

1　4　7……

请说出在正常情况下，下一个数应该是什么？此外，还有哪些数也是正确的？请给出至少三种这样的情况，并分别说明理由。”

PAT 的另一个部分是算术推理，但是没有任何一个人能够告诉我，为什么程序员必须在算术方面很擅长。在 1956 年之前，也许算术能力很重要，因为和现在相比，那个时候的程序开发工作和机器语言的关系更为直接。在使用更为原始的语言开发程序时，你可能需要能够对2～3个 16 进制数的加法进行手工计算，这样你才可以计算出数据区的某个地址。但是时至今日，又会有多少程序员要这样做呢？既然有了计算机，我们为什么还要做算术计算呢？我自己从来就不曾能够很快地把 7 和 5 相加起来，但是我也从来不认为这会阻碍我成为一名程序员。实际上，正是因为在算术方面能力有限，所以我才会对自己的程序、自己所做的核对计算更加谨慎——这会迫使我更多地进行测试。很容易说明，在算术方面的欠缺反而是程序员的一项宝贵财产。

PAT 的第三个部分则更是令我感到困惑。但是，也许我没有抓住要领，即使是在 1956 年，对几何关系的理解力似乎也与程序开发的能力没有什么关系。而在今天，几何方面的能力与程序开发的能力就更扯不上关系了。我还从来没有见过，有哪位程序员需要辨认在旋转 90 度后的两段程序是否相同。（我确实见到过一篇有关回文编程的文章，这类程序无论是按照正向或反向阅读，都是一样的；但是，这类研究对于专业的贡献微乎其微。）

倘若人们非要坚持使用这类测验不可，那么也许现在是应该把一些新思想引入到这类测验中来的时候了。人们肯定会这样坚持，因为虽然使用

这种方法取得成功的可能不大，但是他们由此得到的回报却很大。因此，还是恕我冒昧地就如何才能更有意义地确定在将来会获得成功的程序员，提出几点建议：

1. 为被测准备30～40页的纸，上面的内容是随机的；同时给他提供10个盒子以供存放这些“资料”。过一个星期后，请他找出上面印有单词“COMMODITY”的那张纸。根据其查找的速度，相应地给出评分——最快的人的得分最高。

2. 类似上面所提到的方法，采用诸如想像数列中的例外情况的试题。

3. 告诉他，你正在思考一个他必须解决的问题。除非他明确地询问，否则不要给他提供任何有关的信息，而是按照字面的意思回答他的问题。根据其猜出问题的多少给他评分。对此类试题，我确实不抱多大希望：它们是否会行之有效？即使有效，是否可以应用？——但是，它们至少会比截至目前为止我们所使用的方法更有希望。

不过，事情还不是完全没有指望。当我们在经验丰富的程序员中进行挑选时，情况则迥然不同。但是，极少有作者或雇员能认识到这一点。比如，根据一次对282家使用PAT测验的单位进行的调查结果，在从经验丰富的程序员中进行选拔时，只有138家单位依然使用这种方法。原因何在？因为他们认为自己没有其他选择。既然没有其他更好的办法，他们自然只好去尝试那些现有的方法——虽然尝试这些令人难以琢磨的方法，不过是徒劳之举。只要有人向他们许诺说可以提供一个别致的方法，这些公司就很容易上当受骗——既然到处都是冤大头，那么江湖骗子自然就会大行其道。

这种事情离我们并不很远——比我原先的估计还要近。在准备好一个这样的圈套之后，我就会接着在商业杂志上发出这样一则广告：

测验你手下程序员的技能水平

“本实验名为_____，其中提供多种过程、软件、指导和问题，可专门用以评测程序员的技能水平。_____可以衡量编写正确的、紧凑的和高效的程序的能力，而且既可以被用以测验经验丰富的程序员，也可以用以评测非专业程序员在程序开发方面的智力水平。这些测验与具体的程序语言无关，而且可以根据问题的难度以及被测的解题时间相应给出评分。_____程序运行于（某某某）系

统。定价：5 000 美元……”

作为读者，你可以对这则广告做自行评定。但是，除了其所宣称的可以同时适用于有经验和无经验的程序员之外，其中也确实为设计测验的正确方向提供了些许的线索。比如说，其中提到了“编写正确的、紧凑的和高效的程序的能力”。这段话暗示了些什么呢？

正如有一位作者提到的，如果正在为管弦乐队招聘一位双簧管乐手，我们可以试听一次。这样，也就大致几分钟的功夫，我们就能够确定其是否有资格担任这个角色——至少，这样可以很快地排除掉根本不称职的那些人。接下来，在经过更长的一段演奏之后，我们就能够对该双簧管手的能力有比较恰当的评价。但是，这位作者转而感叹道，可惜对于程序开发来说，“没有与这种试听测验对应的方法”。然而为什么没有呢？对于一个没有经验的人来说，确实没有；但是如果有一个小孩子前来应聘演奏双簧管的实习职位，那么也是同样没有什么测验的办法。

既然有的程序员自称极有经验，为什么不让他们坐下来，写上一小段程序呢？或者，也可以让他们阅读并理解一整套程序要求，然后简要地讲解一下如何加以实现。真的，为什么不呢？也许，这是因为我们太难得去读程序了，所以我们根本没有想到使用这种方法；或者，当我们想要找到“经验丰富的”程序员时，即使只有一名应征者出现，我们也会认为自己很幸运了。但是，对惟一的一名程序员做测验，实际上就像你测验自己的妻子一样——纵然她不能通过测验，你又将如何处置呢？

小结

关于哪些是优秀程序员的重要素质，每一位参与招聘程序员工作的人，都会有自己的看法——尽管我们已经看到，程序开发绝对不是某一个人可以独自完成的。正如此前所引述的，Mayer 相信他提出的“X-要素”与智力有关。对于那些积极致力于心理测验的人们来说，他的至理名言具有警示的作用。人们越是热衷于程序开发，他们就会越发对自己基于印象的认识充满信心。但是在对自己的 X-要素的了解方面，并非

所有人都能达到 E. W. Dijkstra[①] 那样的程度。他曾经这样说过：

> 我参与了研究生层次的教学工作，我们的目标是培养某种类型的“数学工程师”。当需要考察某人是否可以成为我的学生时，我所知道的最有力的测验指标就是：他必须能够纯熟地驾驭自己的母语；为此，你所需要做的就是竖起耳朵去听。

由于 Dijkstra“参与了研究生层次的教学工作”，所以我们不要受到他这番话的影响——因为正如其他几位工作者一样，他可能的确从学术研究的成功中提炼出了某种决定的因素。与**学术方面的**成功一样，“语言能力”也是衡量程序开发能力的很好指标——所以，也许 Dijkstra 的直觉是对的。但是如果我们的目标是商业软件而不是所谓的“玩具”软件，那么就实际程序开发工作的绩效而言，也许除了一般性智力之外，我们根本没有任何方法可以对智力进行测量。就我本人而言，我相信较之智力因素，人格因素、工作习惯以及培训等方面的因素要与此更为相关。这些因素与智力因素不同，它们都可以通过后天经验发生改变。因此，**选拔**程序员的问题就转化成为**培养**程序员的问题。换而言之，优秀的程序员是培养出来的，而不是天生的。因此，我们必须把注意力放到培养或者培训的过程上来。

思考题

请主管回答：

1. 你目前在选拔程序员时，是否使用智力测验的方法？你所在单位的人事部门呢？如果你们使用这种方法，你们对其合理性有多少了解？

① E. W. Dijkstra（1930/05/11－2002/08/06），荷兰数学家，ALGOL 的推广者，率先实现了 ALGOL60 的编译器。他还在图论、算法和操作系统方面有很大的贡献。图论中的求最短路径的经典方法之一就是 Dijkstra 算法。还有操作系统中的 PV 操作（荷兰语通过和释放的意思）也是他提出的。1972 年获代表计算机科学界最高荣誉的图灵奖。——译者注。

为了验证其合理性，在程序员经过一段时间的工作之后，你们是否会对他们进行评估？你们进行评估时所使用的是什么方法？你对这类方法的有效性有多大把握？

2. 通过对手下程序员的观察，你认为他们中最经常缺乏的是哪一种重要的智力因素？为了检测他们是否缺乏这方面的能力，你认为可以采用何种测验？

3. 你是否愿意花上 5 000 美元购买广告中所建议的某一整套测验？要是每项测验只需花 100 美元呢？如果你不愿意，原因何在？如果愿意，你是否读过本书？

请程序员回答：

1. 在你应聘目前的职位时，是否参加过智力测验？如果参加过，你认为测验与工作有关系吗？据你所知，你所在的单位是否还在使用这些测验方法？如果是，而且你认为这些测验与工作风马牛不相及，你会为此做点什么？

2. 你是否认为，至少有某些对程序开发来说至关重要的智力因素，可以通过测验检查出来？请列举出，你所认为的对程序开发来说至关重要的那些智力因素；同时也相应地列举一下，对各种智力因素，应该如果进行测试。

3. 在本章中所提到的问题求解能力或者习惯，是否有哪种是你所缺乏的？如果有，你计划如何改进？

参考资料

- Mayer, David B., and A. W. Stalnaker, Use of Psychological Tests in the Selection of Computer Personnel, *SHARE/GUIDE Presentation*,

October, 1968.

- Reinstedt, R. N., et al., *Computer Personnel Research Group Programmer Performance Prediction Study*, The RAND Corporation (RM-4033-PR), Santa Monica, California, March 1964.

- Biamonte, A. J., Predicting Success in Programmer Training, *Proceedings of the Second Annual Computer Personnel Research Conference*, Association for Computing Machinery, New York, 1964.

- Gotterer, M., and A. W. Stalnaker, Predicting Programmer Performance Among Non-preselected Trainee Groups, *Proceedings of the Second Annual Computer Personnel Research Conference*, Association for Computing Machinery, New York, 1964.

- Okimoto, G. H., The Effectiveness of Comments: A Pilot Study, *IBM SDD Technical Report TR* 01.1347, July 27, 1970.

- Luria, A. R., The *Mind of a Mnemonist*, New York, Avon Books, 1968.
这是一个很好的例子，它说明了在与记忆力有关的任务中，一个人可以如何运用其特殊的能力与不足以取得了不起的工作绩效。

- Hunt, J. M., *Intelligence and Experience*, New York, Ronald, 1961.
Hunt 对智力发展的过程作了分析，最后指出了哪些措施可以对智力产生影响。不幸的是，根据其研究结果，存在一个程序员的年龄上限——在超过这个年纪之后，一个人的智力就几乎不可能再有任何提高了。因此也许我们应该“莫等闲，白了少年头”。现在，越来越多的父母已经开始把终端带回家中，我们培养出精于编写程序的莫扎特的那一天还会很远吗？

- Wertheimer, M., Productive *Thinking*, revised ed. New York, Harper, 1945.
Wertheimer 是 Gestalt 心理学学校的主要创始人。在本书中，根据自己的经验以及对年幼儿童的实验结果，他提出了自己的见解，这些意见告诉我们应该如何教会人们通过思考获得更大的成果。虽然在让程序员们懂得如何更有效编程的过程中，他的这些思想似乎还不能直接加以应用，但是此书确实包含了很多有益的建议，可以让我们了解到哪些是值得尝试的，哪些又是应该避免的。

- Polya, George, *How to Solve It*, Princeton, Princeton University Press, 1946.
Polya 是当今最伟大的数学家之一，在这本畅销书中，他介绍了自己针对数学问题和难题的求解技术的研究。同样地，这本书对程序开发也非常有启发意义。不过，虽然有众多的程序员曾经读过这本书，还没有人曾经对其作过直接测试。

□ Ghiselin, Brewster, *The Creative Process*, Berkeley, California, University of California Press, 1952.

虽然人们对所谓的“创新过程”已经研究了很长的时间——至少可以追溯到柏拉图和亚里士多德时代——但是我们还远远没有真正理解这个概念。我们甚至连很多基本的问题都还不清楚：比如，程序员是否的确需要具备创造力？是否需要莫扎特或者爱因斯坦那样的创造力？是否有不同层次的创造力？或者，只是不同层次的创造成果？但是无论如何，对于本来就猜测“程序开发需要某种创造力要素”的那些人来说，这本原创作品与精彩摘录的汇编专集，无疑会使他们备感振奋。即使对于不相信程序设计需要创造力的那些人来说，也是如此——尽管他们可能没有机会了解过爱因斯坦、莫扎特、梵高、沃兹沃斯①、柯尔雷基②、叶芝③、尼采④、荣格⑤以及斯宾塞⑥等针对这一问题的讨论。此外，这也是本章所提及的 Stephen Spender 那本书（平装版）中排版错误的出处。

□ Sackman, Harold, *Man-Computer Problem Solving*, Princeton, Auerbach, 1970.

Sackman 的研究范围并不是仅仅限于程序开发，而是面向更为一般的问题求解行为。

① Wordsworth William，沃兹沃斯·威廉（1770--1850），英国诗人。其最重要的全集《抒情歌谣》（*Lyrical Ballads*，1798）同柯尔雷基·塞缪尔·泰勒柯尔斯基合作出版，为建立英格兰诗歌的浪漫主义风格做出了贡献；1843 年，他被授予桂冠诗人称号。——译者注

② Coleridge Samuel Taylor，柯尔雷基·塞缪尔·泰勒（1772－1834），英国诗人，批评家，浪漫主义流派的倡导者。同华兹华斯·威廉一起出版了《抒情歌谣》（*Lyrical Ballads*，1798），里面包括了他最为著名的诗歌《老水手的故事》（*The Rime of the Ancient Mariner*）。 译者注

③ Yeats William Butler，叶芝·威廉·巴特勒（1865－1939），爱尔兰诗人及剧作家，被认为是 20 世纪最伟大的诗人之一。是都柏林阿贝剧院的爱尔兰国家剧院公司的创始人；他创作了许多短剧，包括《凯瑟琳女伯爵》（*The Countess Cathleen*，1892）；他的诗作成集出版，如《回旋楼梯》（*The Winding Stair*，1929 年），内容从早期的爱情抒情诗到晚年复杂的象征主义作品；1923 年，他获得诺贝尔文学奖。——译者注

④ Nietzsche Friedrich Wilhelm，尼采·弗里德里希·威廉（1844－1900），德国哲学家。他论断基督教强调的来生说使它的信徒们不能很好地处理现世的生活；他坚持认为理想化的人类、超人哲学（Ubermensch）能够创造性地引导情感而不是被它们所压制；他的著作包括《善与恶的彼岸》（*Beyond Good and Evil*，1886）和《查拉图斯特拉如是说》（*Thus Spake Zarathustra*，1883－1892）。——译者注

⑤ Jung Carl Gustav，荣格·卡尔·古斯塔夫（1875－1961），瑞士心理学家、精神病学家。创建了分析心理学；在了解人类心智方面做出的贡献是，提出了外倾型（extraversion）和内倾型（introversion）的概念以及集体无意识（collective unconscious）的概念；其著作包括《无意识心理学》（*The Psychology of the Unconscious*，1912）和《心理类型》（*Psychological Types*，1921）。——译者注

⑥ Spencer Herbert，斯宾塞·赫伯特（1820－1903），英国实证主义哲学家。他试图在其系列论著《合成哲学》（*Synthetic Philosophy*，1855－1893）中将进化论运用于哲学及伦理学。——译者注

他的这本书中不乏有益的启示，可以让我们看到使用计算机对问题求解行为有哪些作用，同时又缺乏哪些作用。但是在某种程度上，他没有把程序开发当成问题求解来看待。Sackman本人并非程序员，而是一位心理学家，从书中的内容来看，似乎他根本就没有**阅读过开发出来的程序**。与大多数心理学家一样，他明显地做了这样一个假设：同一问题的任何完整答案必然相互等价——因此他认为需要仔细考察的，只有无法完成程序以及得出正确结果这两种极端情况。但是正如我们所知道的，输出结果相同的程序可能不计其数。因此，只有等到心理学家们能够认识到这一点之后，才有可能揭开蒙在大多数程序开发/问题求解行为之上的面纱。

□ Weinberg, G. M., *Experiments in Problem Solving* (Doctoral Thesis), Ann Arbor, University Microfilms, 1965.

如果哪位心理学家对上面的评论不甚理解，我建议他读读这份论文。本论文研究的结果揭示了这样一个问题的答案：对问题求解任务进行的足够细致的观察，以及依照不同个人的长处与短处来描述任务的能力，将如何使我们得出对问题求解的真知灼见？如果你的观察过于细致，或者如果你为了获得所谓“统计意义上的重大研究成果”而抹煞不同人在个性、能力方面的差别，那么你就绝对不可能对问题求解有深刻的理解。

第9章　智力水平以及问题求解能力

评　　注

这一章的来历很有趣，好在今天终于可以说明一下。在过去几年中，不断有人问过我：为什么要在程序员的智力测验方面花费如此多笔墨，而得出的结论却彻底地否定了这些测验的意义？实际上，我在最初的手稿中谨慎地回避了这个问题，而初稿的评委中偏巧有两位就是从事这类测验的推销业务。很自然地，他们要求本书讨论一下“计算机程序开发中最关键的问题”。为了满足最初的出版商的要求，我才撰写了本章中关于智力测验的部分。

时至今日，程序员的智力测验问题已经不复存在。要是在消除这种荒唐做法的过程中，这一章曾经起了一点作用的话，那么我将为自己的工作感到骄傲。是的，智力确实是程序开发取得成功的一个因素——如果一个人的智商在50分以下，估计他不太可能参与操作系统的开发（尽管这些人可能去设计各种新语言——是魔鬼让我这样说的）。

除了智力以外，程序开发的成功（或者失败）还取决于其他的很多因素。我确实曾经发现，有些人貌似驽钝，却可以写出高质量的、很有用的代码；而另外一些人虽然看起来更显聪颖，却从来也没有写出过任何有用的代码。简而言之，今天的我依然坚持自己经过了25年时间考验的立场：

> “……较之智力因素，人格因素、工作习惯以及培训等方面的因素要与此更为相关。这些因素与智力因素不同，它们都可以通过后天经验发生改变。因此，选拔程序员的问题就转化成为培养程序员的问题。”（p. 243）

我所谓的“培养程序员”，并不是强行要求他们按照某种“最佳”模式进行思考。我在自己的博士学位论文中曾清楚地指出：问题求解的模式是因人而异的[1]。所以，如果有什么力量强制某个程序员按照另一个人的模式进行思考，那么就必将会削弱该程序员解决问题的能力。因此，最大的挑战并不在于创造性思维本身，而是创造性的交流，用可以为（各有其独特思维方式的）其他人接受的方式重新表述我们自己的思想。

注释

[1] Gerald M. Weinberg, Experiments in Problem Solving (Ann Arbor, Mich.: university Microfilms, 1965).

第10章 积极性、培训以及经验

在心理学家的眼中，人们在完成特定任务时的工作绩效，是由任务本身及其承担者对任务**理解**的深度共同决定的。心理学家同样认为，工作绩效同时也受到任务承担者个体在诸如人格和智力等方面差异的影响。尽管人格是可以改变的，而且智力也可以有所提高，但是工作绩效的实质提高还要依赖于培训和实践经验。

不过，心理学本身并非一门严密的科学，而且也从来没有严密过。无论心理学家如何对任务及其承担者的理解深度进行考察，无论他们如何对任务承担者个体之间的所有差异进行衡量，而且，无论他们如何考虑到培训与实践经验，工作绩效总还是有很多方面无从解释。例如，同一个人在前后两天分别承担两项任务，即使这两项任务在本质上是一样的，他在后一天的工作绩效却可能会比前一天差很多。由于前后只相差一天，因此可以假定其人格和智力都不会有任何不同；另外，在经过一天之后此人在对任务的理解程度与实践经验方面只可能有所提高——既然如此，这种工作绩效倒退的现象又如何解释呢？

在所有因素都被考虑到了之后，剩下的那些无从解释的部分就称为“积极性”。这方面因素是否存在？尽管心理学家们对此依然心存疑虑，但实际上，讨论这方面问题的专著已经有数卷之多了。通过对开发团队或小组工作绩效的偏差进行分析，我们可以发现“积极性”这一因素的确存在，我也曾就其某些特征作过介绍。就某种特定意义而言，鉴于我们难以给出适当的解决办法，在这里还是先不要去打开这个“潘多拉盒子”。但是在进一步就教育与培训进行深入讨论之前，我们必须首先就积极性问题

作一讨论——这有两方面的原因。首先，既然在工作绩效中确实有某些部分需要借助积极性这一因素来解释，那么这些部分也就无法通过培训与教育进行解释，因此通过后一因素所能获得的效果必然是有限的。其次，这些无法解释的部分也可能只是教育和培训而已。经常出现的情况是：如果某个人的积极性没有激发出来，就不可能很容易地使他投入到学习之中；反之，如果他动力很足，就没有任何办法可以阻止他学习。

积极性

当我翻开一本标准的心理学教材中有关积极性问题的那一章的开头，下面这段话就会印入了我的眼帘，于是我一下子就领悟出了关于积极性的最佳描述说明：

> 即使是在最匆忙间对人类的行为作一考察，我们每个人都会同意这样一个观点：人，并不只是被动地受制于环境的玩偶。相反地，人们更愿意相信：人类似乎可以自行确定应该对环境中的哪种刺激做出反应，哪些是应该忽略的，哪些信息是值得学习的，哪些信息又是可以置若罔闻的。这些观察的结论，构成一种理论的基础。这个理论可以用来解释人们的某种内在的“导向驱动力”或者“内在的弹力”。这种内在的导向力，正是我们大多数人所指的“积极性”。（Krech，Krutchfield and Livson，1969）

这段文字引发了一个有趣的启示，原因在于：这并非一本新书，而是购自于某大学书店的一本二手书。这本书的前任主人在这段文字处作了些标记，这些标记是用桔红色的铅笔在不同单词的某些字母上小心翼翼地标明的。很明显地，这都是些涂鸦之作；而其作者当时或者正在学习，或者正在课堂上听讲。如果按照顺序把各个字母读出来，结果就是：

S－A－L－L－Y－S－A－L－L－Y

对于“我们大多数人用来解释‘积极性’一词”的那种“内在导向力”而言，还能有什么比这更好的说明呢？

当然，我们没有必要指出，程序员也许有时同样会想知道 Sally 出了

什么问题；而实际上，与此同时他们的主管可能更希望他们正在考虑计数器[①]出了什么毛病。但是，主管们所犯的一个错误就是，他们总是假设：工作绩效差的原因是由于缺乏积极性。于是，他们会企图借助一点“外部驱动力”来弥补“内在驱动力”的不足。而在这种时候，其实程序员所承担的外部压力往往已经够大了，而并非不够。

关于积极性，最广为人知、也最广为人接受的一项研究成果就是：如果适当地增加“驱动力”，那么在开始阶段确实可以提高工作绩效；但是一旦超过一定的极限之后，继续增加这种“驱动力”只会很快地令工作绩效降为零。在复杂的任务中，更加容易看到这种工作绩效急速下降的现象，这正是它对程序开发如此重要的原因所在。比方说，过于努力地去查找错误，与根本不去查找同样坏，甚至还要更坏。实际上，往往要等到程序员已经决定放弃从而不再承受压力之后，许多程序错误才能被排除掉。给程序员施加高压，以期他们能够很快地排除某个程序错误，这种做法已经被证明是最差的策略——尽管截至目前，这仍是最为普遍采用的策略。

因此，无论是从程序开发主管还是程序员本人的观点来看，关于积极性，最重要的问题就是：我们对程序员的激励已经达到了何等程度？根据这个问题的答案，我们才能确定是要增加驱动力，还是减少。根据我的经验，总的看来，如果多名程序员组成一个集体，那么他们往往会被激励得过了头——如果你要问，随着压力的增加，为什么有那么多的开发项目小组最终会土崩瓦解，那么这就是一个主要的原因。

但是，即使已经确定了是应该增加或是减少驱动力，我们又将如何具体操作呢？任何个人所面临的问题似乎都是无法解决的——因为如果不是自觉地去做事，如何才能改变自己内在的驱动力呢？尽管如此，实际上程序员也可以采取其主管所采取的策略——只有首先改变其所处的外部环境，才能有可能去改变自己内在的一些东西。

当然，外部环境中特定某方面的改变，对开发组内不同成员的内心所造成的影响也不会完全相同。如果让一位裸体女郎从门前走过，那么她对男人与女人的影响肯定有天壤之别。而且不同的男人或者女人所受到的影

① Tally，计数器。这里，作者利用 Tally 一词与 Sally 在拼写、发音上的近似，将讨论从一个具体的事例，巧妙地转到计算机程序开发上来。——译者注

响也不尽相同——关于这个问题，那部偷拍的影片“What Do You Say to a Naked Lady?”[①] 已经作了活灵活现的说明。但是，组成开发团队的程序员们在对外界刺激的反应上，也许还有一些共同之处。正如男人们一样，无论他们之间的差异有多大，面对裸体女郎的反应一般还是有些相同的地方——至少，在对感兴趣方面的反应上，确实如此。

关于外部因素对程序员有哪些促进作用的问题，我们已经进行过一些预备性的研究，希望得到一些有关认识。我们没有通过实验来开展这些研究；那样的话，我们必须对不同的程序员施加不同的外部刺激，然后观察其相应的工作模式。实际上，我们只是对程序员及其主管进行了调查，以了解他们**认为**“什么才是促进程序员工作的动力来源”。在这里我们强调了“认为”一词，因为人们口头上声称的正在或者将要做的事情，与他们实际的所作所为之间没有任何必然的联系。在做了这种保留之后，我们就可以介绍一些更加有趣的发现。

在对一个大型软件开发基地的软件程序员的调查中，我们发现，“涨工资和/或发奖金”将首先会“令我的主管觉得耳目一新，因为这会使我持续高效地工作，或者至少会使我勤勤恳恳地工作”。正如根据社会心理学理论可以预料到的，紧随其后的就是“个人在任务规划中的参与”。而并列第三的情况有两种——“职位的提升”以及“对我的工作质量给予更多的关注”。

在所有的 19 个选择中，这 4 项名列前茅。位于最后的选择是“缩小我工作的范围”——这很合理，因为这一选择与“更多地参与整体规划”

① 由 Allen Funt（1914－1999）在 1970 年（本书写作前后）自编、自导、主演并制作的一部影片。Funt 因主持一个名为“偷拍镜头（Candid Camera）”的著名电视节目而闻名，并被称为偷拍先生（Mr. Candid Camera）。

虽然这实际上只是一部喜剧，但是在当时却被打上了色情的标签，以至于没有哪家电影院敢于放映。在这部影片中，Funt 用偷拍的方法记录了普通美国人在公众场合面对裸体的不同反映。其中的一段，与本书第 6 章“绩效评价”一节所介绍的心理测试有异曲同工之处：Funt 通过广告“诚征”别人前来面试。当应征者被带到 Funt 的办公室时，他看到那里还坐着很多其他的“面试者”。这些人实际上都是 Funt 的同谋，一旦见到 Funt 的暗号，他们都迅速地脱得精光。真正的面试者首先是猛地一愣，但在对自己的处境稍加审视之后，他很快就“领悟”出了些什么——他也从座位上站起来，同样迅速地脱得一丝不挂。他不仅没有打听一下事情的原委，反而站在那里满脸期望地看着周围的人，等待着他们下一步的“集体行动”。——译者注

似乎是吻合的。在此选择之上，是另外 4 项不受青睐的选择。首先是“在文档化、复印等方面的支持”。这一结果与世人普遍认同的“文档化工作简直就是灾难”的观念表面上互相“矛盾”，但是做出这样的结论还为时尚早——可能是因为那些人已经积累了一定的资历，以至于他们早已不用亲自动手去做文档化工作，于是对他们来说，这方面的因素已经不再重要了。这 4 个选择中的第 2 项是“被分配到受人尊敬的岗位”，这一项似乎与“职位的提升”相矛盾。再进一步的调查显示，“职位的提升”归根结底总要落实到“收入的提高”上，而不仅仅是“获得什么新的头衔”——这显示出进行观念调查的困难性。

这 4 个选择中的后两项看起来有点互相矛盾：“放宽预定的目标完成时限”与“紧缩预定的目标完成时限”。但是，只要注意到，**在这个时候**，预定的目标完成时限对于这个开发组来说无关紧要，那么我们就可以解释这种矛盾。在这个时候——是一个重要的限定词。在进行本次调查之时，离项目预定的完成期限还有至少 6 个月。我们很希望能在几个月之后再检查一次，来看看“预定的目标完成时限”是否还是位于排行榜的末尾。

这给我们针对积极性问题的研究造成了另外一种主要的困难。即使上述调查中的意见确实能够反映外部力量的强度，也只能说是反映了在一个特定时间、一个特定环境下的情况。在项目开始之初，紧锣密鼓地进行的只是目标设定的工作，而此时离预定的目标完成时限还很远；但是随着项目不断朝最终目标推进，人们受到的激励将与项目开始之初时有所不同。任何人在参加工作之初所受到的激励，与后来也不尽相同——比如，随着存款的增加、工资的提高以及个人负债的逐渐还清，金钱的重要性会不断降低。另外，正如我们在某些特定环境中所看到的那样，额外的金钱实际上反而可能会让某些人放弃自己的工作。

也许你现在会说：金钱的激励作用还不如提高对目标设定的参与程度和对工作质量的关心程度来得实在。这种结论似乎显而易见，因此你可能会轻易相信：在工资水平相当的前提下，与工作性质不同的其他群体相比，程序员群体会更不在乎金钱。但是，这种观点实际上有失偏颇。当然，程序员们的工资标准相对一般的老百姓通常要更高，所以金钱对他们的激励作用更小就不足为奇了。情况也许确实如此，但是我们的研究并

不能证明这一点。而且同样地，金钱是一种模棱两可的激励因素。对有的人来说，金钱的激励作用表现为一种手段，借此可以获得自己想要的或需要的东西；而对另一部分人来说，金钱的激励作用则不过是一种显示其工作价值的象征。但是，无论具体有何差别，金钱毕竟就是金钱，难道不是吗？这句话并不确切，因为要是金钱的价值不仅仅是作为一种象征物，那么为了激励程序员，就可以授予他们金钱之外的奖赏——比如说，对他的工作质量给予更多地关心，或者让他负责一点规划工作。需要提醒你的是：这并不只是要为公司节省经费，而是为了更有效地激励他们；而涨工资只是权宜之计——当前价值的标志并不在于工资的绝对数值，而在于其**增长量**。从长远来看，持续不断地涨工资恐怕是不大现实的；然而，持续不断地让程序员参与任务规划和持续不断地关心其工作质量都是不成问题的。

与工作中的程序员有所接触的任何人——比如程序员的妻子或者丈夫，可能在自己的丈夫或者妻子正在运行某个程序或者正在参加一次关于程序错误的讨论时，曾经打断过他们——都会懂得：只要允许程序员按照自己所偏好的方式进行工作，那么程序开发这项工作本身就是对他们最大的激励。有一句关于象棋的古老格言是这样说的："正如女人和音乐一样，象棋也具有令男人感到愉悦的魅力。"[①] 我在此斗胆略作修改："正如象棋、女人和音乐一样，程序开发也具有令男人感到愉悦的魅力"。我想，有朝一日我可能会拾到一本书，却发现正是在下的这一拙作，而有人已经用笔圈出其中的若干字母，它们按照顺序读下来是：

F－O－R－T－R－A－N－F－O－R－T－R－A－N。

① 这句格言出自 Siegbert Tarrasch（1862/03/05－1934/02/17），德国医生和著名国际象棋选手。他虽然足部先天畸形，但是智力超群，被公认为 19 与 20 世纪之交国际象棋的"无冕之王"。在他所处的年代，由第一位国际象棋世界冠军 Steinitz 所创立的正统理论认为：一盘棋的正常结局应该是和棋；因此，只有等待、发现或诱使对手发生（无论是防守还是进攻上的）错误之后，才可能通过攻击获胜。但是 Tarrasch 却独树一帜地认为：攻击本身可以带来优势，使对手需要防御而后退；而通过控制中心、保持子力通畅与加快进攻速度（有时为此可以不惜出现孤兵等阵形缺陷，甚至牺牲子力），可以令对手防不胜防——在他的无数精彩对局（比如 1914 在圣彼得堡战胜 Nimzowitch 的弃双象局）中，这一思想体现得淋漓尽致。Tarrasch 在国际象棋方面的著述颇丰，此处的这句话出自其最后一本（也是最经典的一本）著作 "*The Game of Chess*"，原句为 "Chess, like love, like music, has the power to make men happy." Weinberg 在此处的引用与原文略有出入。——译者注

培训、课程学习与教育

有些人声称，随便在自己门口揪住一个过路人，他们都能够把这个人培养成为一位名副其实的程序员。因为并非每个走过计算中心门口的人都是来求职的，所以这一观点未免有些失之偏颇。尽管如此，他们的话中还是有些许的可取之处。曾经有一次，在造访一家获得巨大成功的软件公司时，我们有幸整个下午都与他们的系统程序开发团队呆在一起。其中有 5 个人都担任了要职，而他们相应的资历如下：

1. 一位年纪 33 岁的男子。他在大学二年级时被常春藤联合会学校[①]开除，因为他当时惟一想做的事就是打桥牌。

2. 一位年纪 28 岁的妇女。她曾就读于南部的一所私立女子学院，当时的专业是西班牙文学；但是此后她沉迷于花天酒地的生活；这样整整过了 3 年之后，在一项旨在解救吸毒者的计划的帮助下，她才得以新生——她的程序开发知识，正是在此期间掌握的。

3. 一位前计算机维护工程师，年纪也是 28 岁。他在小学之后所受过的正式教育，仅仅是进入过海军电子学学校。

4. 一位有 5 个已成年孩子的母亲，年纪 55 岁。35 年前她在师范学校获得过学位，但是由于随后就结婚了，所以根本没有从事过教学工作——而当她的孩子都长成并离开她之后，她才打算用程序开发来打发时光。

5. 一位年纪 30 岁的男子。在（以优异成绩）大学毕业之后，他直接攻读数学硕士学位，于是他就进入了程序开发这一行业。他在团队中被认为是一个性格古怪的人。

这个开发组中成员的背景千差万别，由此看来，似乎计算机程序开发培训的必要性值得怀疑；但是实际上，这些程序员的情况所推翻的，只不过是将培训的基础建立在“正规”教育之上的观念。该开发组中各成员的共同之处只有一点——他们在同一个人的手下共同工作了 4 年，而这个人

① Ivy League，常春藤联合会。由美国东北部八所大学和学院组成的一个联合组织，包括布朗大学、哥伦比亚大学、康奈尔大学、达特茅斯大学、哈佛大学、普林斯顿大学、宾夕法尼亚大学和耶鲁大学。——译者注

正是该软件公司的创始人之一。在这 4 年中，这位主管按照自己的构想成功地改变了他们。

虽然这种情况听起来似乎不寻常，但实际上并非如此。只要在某种程序开发的环境中呆过一段时间，大多数人都可以举出自己曾经遇到过的、类似的其他开发组。这类开发组最显著的因素总是相同的——成员们都在一位令人敬畏的领导手下，为一个实际的项目共同工作过数年。然而，这数年间的时间意味着培训与经验的积累。尽管或许这些培训与经验积累不是我们通常或常规意义上所讲的那种，但是，只要我们抓住这一过程的本质，就同样可以达到这一目标。

在教育发展的过程中，人们在观念上存在种种混淆。其中之一就是，将“参加课程学习”与“接受教育”混为一谈。另一个容易混淆之处是，“接受教育”与“参加培训”又有什么不同？为了区分这些混淆之处，进而说明为什么它们会妨碍人们的进步，我很乐意提供关于自己的教学工作有效性的一些结果。在多年来讲授操作概念的课程中，我发现，要是学员缺乏操作系统的上机经验，那么这些概念的讲解就会显得空洞无力。其结果是，在我的学生离开学校大约 1 年之后，当我前往其工作现场进行访问时，却发现他们根本没有用到在我的课堂上所“学到”的东西。

学习过程中为什么会出现这种损失，我将其原因归结为缺乏实践经验。因此，我曾决定用 OS/360 专题讨论会来代替原来的课程。但是在专题讨论会的模式中，我又发现，学生们在深入了解这些概念时非常吃力。尤其令他们头疼的，是任务控制语言（Job Control Language—JCL）概念的细节——这一现象与来自这一领域的研究报告结果非常吻合。更为特别的一点是，如果可以将学生的领会程度作为一种衡量指标的话，那么在对“无足轻重的”空格以及附加卡片的处理方面，学生们会遇到非同寻常的困难。

于是我决定，在演讲中对这些细节咬文嚼字地逐一强调，并且通过课堂提问，不断跟踪学员们的学习效果。课堂上的这些提问覆盖了一系列有关任务控制卡片的生成、穿孔以及测试的技术，其难度随测验的进行不断提高。但是其结果（如图 10－1 中的“班级＃1”所示）却令人沮丧。在所有的 JCL 错误中，有 83%依然是“空格”错误——这正是我在课堂上特意

反复强调的那类错误。

接下来，为了对其学习效果进行比较，我还在另外一个培训班上进行了这种试验——但是不同之处在于，我在课堂上不再花时间去深入讨论“空格”错误，而只是被动地回答某些特定的问题。这个班的行为如图10-1中的“班级#2”所示。通过这张表格可以看出，我在前一培训班的课堂上费了九牛二虎之力来讲授JCL语法的努力，其净效果只不过是使错误出现率从87%降到了83%。从统计意义上讲，这实在**不是**什么了不起的结果。但是，对于研究程序员教育的心理学家们来说，这个结果的意义却是极不寻常的，因为它显示出，“课程学习”——在这里是参加我所讲授的课程——可能与“接受教育”毫无关系。当然，这也可能是因为别的教师对此方面的学习效果也有影响——我可从来也没有以“世界上最出色的教师”自居过。然而反过来也可以毫不夸张地说，我绝不是世界上最蹩脚的教师。因此，在讲解其各自的“知识点”时，肯定会有相当多的教师会遇到这一相同问题。

		问题1		问题2		问题3		所有3个问题		
班级	人数	JCL错误	空格错误	总共	空格错误	总共	空格错误	JCL错误	空格错误	%
#1	29	63	49	101	88	103	84	267	221	83
#2	17	51	42	86	78	105	88	241	208	87

图10-1　任务控制语言初学者的错误统计[①]

既然如此，还有什么必要去向学员们解释这些平凡的知识点呢？很不幸，尽管在这类知识点中的智力成分不大，但是如果没有经过这方面的学习，那么在继续深入学习操作系统使用的过程中，学员们就必然会遇到障碍。这样，就引出了我们需要探讨的第二个比较——“接受教育”与“接受培训”。大致上讲，我们所说的“教育”，指的是掌握一般性的原理与技巧的过程；而我们所说的“培训”，不过是学会某种特定技能的一个过程。

① 原书中此表中的数据有微小的误差。可惜作者已经找不到原始数据，无法进行核对。好在误差并不大，对作者的结论没有任何影响。——译者注

所以，我们可以说“驾驶培训”或者“军事培训”，也可以讲“成人教育”或者“文科教育”。不过，学员在校期间到底会学到什么，不见得总是与这些名词一致。比如，军事培训也肯定会传授某些一般性原理，而且这些原理必将对学员一生的观点都有所影响；而反过来，成人教育也常常仅限于对某些特定的技能进行培训。

然而更为重要的一点是，如果没有前期**培训**的铺垫，后面的**教育**就无从谈起。在我们这个例子中，为了能够不断地学习操作系统方面的概念，学员们就需要具备计算机方面的经验；而为了获得操作系统方面的经验，学员们就必须能够驾驭 JCL。因此，如果在 JCL 语法方面的**培训不过关**，那么在接受有关操作系统原理的**教育时**，也注定要**失败**。

也许，我们应该更加准确地说：后面的教育之所以会失败，是由于前面“JCL‘课堂学习’的失败”——因为正如在图 10－2 中所显示的，班级 2 的学员最终都通过了 JCL 语法的培训。在经过了 10 次练习之后，17 名学员中有 13 人已经对生成任务控制卡片驾轻就熟，不再会出现任何“空格错误”（尽管其中有两名学员在第 11 题上略微出现一点退步，而这道题确实比其他各题要大些）。然而，剩下的那 4 名学员在这种能力上没有什么明显的进步，因此这一阶段学习的不过关，必将成为他们后面学习操作系统概念时的拦路虎。虽然，毫无疑问，他们迟早还是会掌握这方面的技巧，但是课程就在这时嘎然而止，而他们则要卷起铺盖回家了。接下来很有可能的一种结果就是：这 4 名学员将继续对操作系统敬而远之，而另外的 13 名学员现在就可以在进一步的学习中，满怀信心地运用这些知识。

因此，总而言之，在掌握 JCL 语法技巧——接受培训——的过程中所遗留下来的问题，将成为一道障碍，阻碍学员们继续接受有关操作系统的教育。这种遗留问题顺带造成的另一个后果就是，形成了一个由 JCL“专家”组成的小组，而其他程序员依然会像奴隶一般地依赖这些“精英”。仅仅在讨论中对这一问题纸上谈兵，绝不可能消除这种障碍。正确的做法应该是：把学员放到一个实际环境中，让他们通过计算机不断地获取自己所需要的经验。毫无疑问，我们可以建立一个能够比 OS/360 本身更为高效而有效地完成这种任务的系统。比如，这个操作系统可以提供一个支持

对 JCL 语法进行检查的分时终端。然而正是由于 OS/360 的不足，如果我们能让学员来使用这种系统，那么它就会成为一台适宜的教学用计算机。在这种情况下，这正是“课程学习”可以提供的一种服务——鼓励学员们面对面地不断去尝试 OS/360（有一位学员甚至因此戏称之为“障碍系统/360”）。

问　题	JCL 卡片数	空格错误数	平均每张卡片上的错误数	没有犯空格错误的人数
1	85	42	0.50	0
2	136	78	0.58	0
3	170	88	0.52	1
4	136	63	0.46	2
5	136	58	0.43	2
6	102	30	0.29	7
7	153	49	0.31	5
8	170	47	0.28	8
9	85	17	0.20	13
10	119	19	0.16	13
11	187	39	0.21	11

图 10-2　参加培训的任务控制语言初学者出现的错误统计

学习的阻力

令人吃惊的是，只有当存在阻碍学习的负面作用力时，学习才会失败。之所以说“令人吃惊”，是因为在教学过程中，人们过分强调了，如何通过各种精巧的措施或窍门去激发学员的学习积极性。但是，只要去观察一下儿童的学习过程我们就会发现，只要任由儿童自己来决定学习的方向，他们总是可以学到大量的东西——虽然这些东西不见得一定是我们所期望的，但是不管怎样，他们将会学到很多。为了能使他们按照我们的意愿进行学习，我们会建立一些“围墙”，然后开启几扇“小门”。如果儿童

依照我们指定的方向去学习，我们就会暗自庆幸；但是即使离开我们，没有我们刻意设定的那些障碍，他们仍然可以学习，而且也许会学得更好。

然而对于成人来说，阻碍学习的障碍通常都来自我们自己的内心；而一旦可以任由自己做主，通常的成年人都不再能学到什么新的东西。首先（而且也是最重要）的一点是：在开始学习之前，我们需要承认有一些东西值得去弄清楚，而自己还不懂。在专业程序员的观念中，承认自己某个方面知识的缺乏，就意味着地位的降低——除非他能够认识到：作为一位名副其实的专业人员，作为一个真正有实力的人，承认自己的不足，不会有任何的损失。

有些人总是心甘情愿地承认自己的不足，而不去尝试学习任何新东西。其原因在于，他们总是想当然地认为，自己不可能取得成功。这种对失败的畏惧心理，有的是由于缺乏自信心造成的，也有的是源自先前类似努力过程中的失败经历。但是，更普遍的情形是，这类恐惧主要地不在于失败本身，而在于失败可能被周围人看到。

如果身处某个集体中，人们更愿意应用自己已经掌握的知识，这样就会妨碍接受新的知识。如果要进行学习，我们首先必须做好犯错误的思想准备——但是如果周围有别人在场，那么克服自己的畏惧心理将会显得更加困难。事实确实如此，如果使用的知识**是众所周知**的，那么也许正是由于有周围人在场，我们会努力去**避免**犯错误。如果把这种结论推广到学习程序开发技术的过程中，我们就可以自然而然地猜测：在首次学习某种新程序语言的时候，如果是在一个只有终端相伴、与世隔绝的环境中，那么学习的效果将会最好——因为这时任何错误都不至于为人知晓。

在很多方面，我们的观察结果都验证了上述的猜测。为了确定在使用PL/I语言过程中有哪些错误是最常见的，以及编译器能够在多大程度上检测到并更正这类错误，我们曾进行过一项研究。在这次研究中，我们收集了来自某个开发组的50位初学者**所有**的程序运行记录。然而，在收集这些信息的过程中我们所遇到的困难之大令人震惊。他们用尽了种种借口，以掩盖确凿的错误记录。我们曾反复明确地在开发组内解释说，我们对这些信息的研究并非针对某个人，研究的**目的**只是为了对所有的错误进行统计测量。然而，我们还是不断得到这样的回答："……可是你们并非

对这一项运行记录感兴趣呀；我已经删掉了那个倒霉的分号。请稍候一会儿，让我再运行一次；我将会给你一个更好的结果。”

还有一个现象也可以说明这个问题：初学者们通常会对终端系统表现出异乎寻常的热情；而等到经过一定时间的学习，他们的水平有所提高、不再会出现这类琐碎的语法错误时，这种热情又常常会一落千丈。更具经验的程序员对终端的热情似乎不高，对他们来说，终端只不过是一台数据输入设备——其意义也许只在于它具有强大的语法检查功能，可以帮助他们消除键盘输入错误。然而对于初学者或者业余程序员来说，他们在诸如语法、键盘输入或者诸如此类的附属任务控制操作方面的困难，要比来自于语义的微妙之处或者问题本身的更深层次的困难多得多。但是，将终端系统与批处理系统进行对比的那些心理学实验，大多数仅仅涉及到初学者的或非常平凡的问题。因此通过这些实验，我们无法获得有关这两种方法的优势对比的确凿结论。如果我们在设计一个操作系统时，希望它将来既适宜于入门者学习，又适宜于熟练者使用，那么我们在对终端的使用情况进行仔细地观察分析时，就必须既考虑到处于不同开发阶段的程序员，也要覆盖着眼于不同类型问题的程序员。

在关于程序语言以及操作系统的多样性的争论中，很少能触及到“便于学习”与“便于使用”二者之间的区别。依长期的眼光来看，最易于学习与最易于使用这二者之间并没有任何必然的联系。在这里至关重要的一个概念，就是所谓“技术的**可扩展性**”——也就是说，某项技术在初次学会之后，还能否在其他环境条件下得到应用。每当我们掌握了一种通用的技术之后，我们的能力并不是简单增加，而是成倍地增加。

还有一些技术，与通用的、可扩展的技术恰恰相反，我们称之为“穷途末路的”（Dead-end）技术。比如，为了求解包含 3 到 4 个方程的系统，需要求解联立线性方程组，为此我们可能会去学习 Cramer 有关行列式方法的规则。事实上，这不过只是高中代数课程中的普通知识。但是，尽管 Cramer 的规则在理论上完全是普遍适用的，但是在实践中却根本不可能在大规模方程系统中得到应用——原因在于，随着行列式规模的扩大，其计算复杂度将极快地提高。在何等规模之后 Cramer 的规则才会不切实际？虽然这个问题没有任何界限明确的答案，但是毫无疑问，它确实是一

项穷途末路的技术。

在程序开发方面，我们时常会碰到，某个程序员，他在解决规模不超过某个特定范围的问题时干得很不错，但是他却无法逾越这个规模界限。实际上，有时我们也会遇到截然相反的情况——有些程序员平时可能默默无闻，直到需要解决更大规模的问题时，他们才有英雄用武之地。在前一种情况中，在小规模问题上大显身手的技术，将无法在大型问题中得到有效的应用；而在后一种情况中，用功能强大而且通用的技术来解决非常简单的问题，只能造成巨大的额外开销。

例如，让我们来对比二分搜索与顺序搜索之间的差别。当编写的程序需要在一个只包含 10 项记录的列表中进行搜索时，如果采用二分搜索，则可能反而会延缓程序的开发进度，甚至会降低程序的运行速度。而在解决这一特定问题的时候，那些也许根本不懂得二分搜索的程序员，可能反而会令那些懂得这种“高级”技术的程序员黯然失色，因为后者虽然掌握了这种高级技术，但是却没有进一步意识到这种技术的应用范围。无论如何，问题的规模翻倍后，二分搜索的速度绝不至于降低一倍，然而顺序搜索则不然——问题的规模稍微扩大（比如在 1000 个记录中进行搜索），其速度的下降就远远不止这种程度。

对于那些已经精通了某些穷途末路式技术（或者程序开发语言）的程序员来说，他们会对自己当前的情况非常满意；所以一旦需要学习新的知识，他们就必须首先（至少是暂时的）克服这种自我满足的心态。在应用新技术的初期，他可能不会获得很大的成功。从某种意义上讲，他们需要使自己的心态重新回到与初学者平等的起跑线上。在学习一种新的程序开发语言时，这种对可能出现的损失的畏惧心理尤其突出。只要你曾经在异国他乡呆过，对那种因为第一次体验到自己在语言知识方面的欠缺，而产生的无助感觉，你就一定有过深刻的体会。而在需要学习新的程序开发语言时，你的那种感觉与这种体会别无二致。在发现自己的法语还不如当地的小孩子时，身处异域的旅行者可能会感到很窘迫；而当必须向实习生请教有关 PL/I 的简单语法问题时，精通 FORTRAN 的程序员也很可能会感慨自己风光不再。

无论是在正式的课堂环境还是工作环境中，对新事物的畏惧、对失败

的担忧以及不情愿承认自己的弱点，都会成为学习的直接障碍。其他的问题则是间接出现的，大部分这类问题的原因，都在于对要学习什么、如何去学习等问题缺乏足够的认识。在一门程序开发的介绍性课程中，任课教师注意到，虽然布置的上机实验题都非常容易，但是大多数学生在完成这些题目时却会遇到困难。学生们的任务仅仅是通过一个连接到 APL[①] 系统的终端，完成两至三道简单的习题；但是经过 5 个星期之后，在 50 名学员中只有 5 人能够提交结果。

任课教师于是决定深入到实验室中进行调查，而他发现有两类不同的问题，最终都导致了一个相同的结果——学习进展缓慢。前一类问题出现在那些到终端室去的学生身上。通常，一旦学会了如何登录并且刚刚知道敲击键盘，他们就会马上去上机。但是他们对自己的能力过于自信，所以总是白白浪费一个又一个的机时，最后总是无功而返。然而后一类问题则恰恰相反，它们出现在那些**不去**机房的学生身上。这部分学员的情况是，他们更习惯于以手工的方式来解决问题，所以除非已经搞清楚了问题的每一个细节，否则他们都会对上机有畏难情绪。然而反过来，如果没有终端的帮助，他们又如何才能把问题的每一个细节都搞清楚呢？

如果遇到的问题来自于自己不熟悉的某个背景，那么通常我们可能会犯两种错误——或者把问题的难度估计过头，或者把问题想像得过于容易。一位蹩脚的老师之所以蹩脚，通常并不是由于对自己所讲授的内容不够精通——在课本或其他材料中，这些讲授的内容是再平常不过的了，学生完全可以在课堂之外自己去细读。恰恰相反，蹩脚的教师实际上可能会在备课时花费更多的精力，以确保其讲述过程的连贯和顺利。这种做法的结果就是，学生们无法看到在老师所举的例子背后，还存在有哪些困难，于是他们自然地就会误认为课程的内容很简单。而在另一些时候，任课教师又可能会刻意地设计一些简单的例子，以强调某些方面。而这种做法同样会导致不利的结果，因为这会给学生们造成一种错误的印象：实际工作中遇到的问题总是这样整齐划一。

比如在我们的 APL 课程中，任课教师会给学员们演示一段用来在……

① APL 语言，是支持远程终端的一种程序语言，具有处理数组的特别功能；该语言具有独特的语法和字符集，主要面向数学应用用户，尤其擅长文字数组的数值计算。——译者注

个字符串（母串）中匹配另一个字符串（子串）的程序——从本质上讲，也就是相当于 PL/I 语言中的 INDEX 函数。在课堂上演示的这段程序实际上只有一行，因此显得非常优美。但是很不幸，这个程序在很多方面都存在缺陷。例如，倘若两个字符串中任意一个只包含单独的一个字符，或者甚至根本没有字符，该程序就将彻底无效。又如，要是子串碰巧与母串首尾连接后的相连处一致，那么该程序居然也会报告“匹配成功”——比如在母串“ABCX”中查找子串“XA”时。该程序忽略了很多需要考虑的细节；但是当学生们需要坐到终端前实际解决这样一个问题时，即使他们不希望看到各种细节问题，也只好硬着头皮认真地逐一处理。

另一方面，这位任课教师也受到了来自相反方向的批评：有人指责他在给学生讲述时，把讲述的内容搞得过于复杂了。比如在某节课的末尾，这位老师发现还多出 10 分钟，于是他就打算给学生们介绍一下自己在研究工作中正在解决的一个问题。但是问题的解答完全超出了一些学生的理解能力，因为这些学生的数学功底欠佳，他们甚至连问题是什么都没法搞清楚。这部分学生因此受到巨大的打击，以至于连课后作业都不敢尝试去做一下。

如果当时有一位教师在场，我们希望他能够对这种情况有所认识，并且进行必要的干预，以使这些学生能够重新回到继续学习的轨道上来。但是，程序开发方面的许多学习过程都没有教师；至少，没有任何人被正式地指定为类似的角色。而且，程序开发方面的许多学习过程也可能在缺乏某些关键要素（比如教材、参考资料、软件以及运行软件所必需的硬件、学习新问题的充裕时间、针对产品问题尝试新技术的授权以及其他可能更有经验的程序员）的情况下进行的。这些要素中缺少任何一种，都会使学习的问题复杂化，并且使得学员在初次遇到令人沮丧的信号时更容易中途而废。

人们通常都会认为，我们在学校学习期间的作用，就是获得这些要素。虽然我们无从知道，一个学校的成功，在多大的程度上更加依靠这些因素，而不是其中的教师呢？但是可以明确的一点是：在我们的教学体制中，教师是最重要的组成部分。由学校来为我们提供这些要素，有何不妥？尽管没有人能够否定这一点，但是存在众多学校这一事实本身就说

明，我们正是在那里才获得了这些要素。主管的推理思路可能会与此类似："他向我申请一个任务编号，但却是为了运行一些练习性的问题。他为什么会这样做呢？我们已经把他送到学校去学习过整整 4 个星期——那里的老师到底教了他一些什么？"

当然，这种态度——"学校是你学习的地方；而办公室则是你工作的地方"——并不只限于程序开发这一领域。只不过与所有其他的领域相比，在程序开发领域对不断学习的要求更加突出，因此我们对其中的那些因素如此重视也就不足为过了。

如何学习程序开发

如果一名程序员想要不断提高，他就不能仅仅依赖于正式的培训；也不能指望主管经理某天大发慈悲，根据需要把他送去进修。同样地，他也不能依靠纯粹的所谓"经验"，因为经验不见得一定会交给他所有的东西。如果程序员需要发挥其经验的作用，他首先就需要学会如何去学习。

掌握学习之道的第一步，就是要了解自己拥有什么、缺乏什么——也就是要有"自知之明"。与坐在教室里的学生相比，自己做自己老师的人拥有一项巨大的优势：他可以根据惟一的学生——也就是他自己——的需要，对课程内容作精确的裁减。在上一节中我们曾经介绍过一个例子，其中的教师发现由于两种原因，自己的学生没有学好 API 语言。其实，这两种原因不仅不同，而且正好是互补的。如果为了帮助前一类学生，这位老师尽力去加深课程内容的难度，那么那部分本来已经觉得跟不上老师的讲解实例的学生，就会被抛弃掉；如果朝相反的方向努力，也不会有什么更好的结果。于是，这位教师就似乎面临着一个令人尴尬的悖论。而实际上，为了走出这种进退维谷的境地，他应该采取的是一种兼顾两个极端的策略——也就是说，他应该根据不同学生的特殊情况，分别进行处理。

而对那些自学的学生来说，如果他有自我感觉能力，那么这种个性化的关注恰恰就是他能够自动获得并进而加以应用的。比如，所有的现代教育理论都对**理解过程中的偏好模式**有所共识。例如，有的人要到课堂去用耳朵聆听，记住（学会）所教信息的效果才会达到最佳；而另外一些人则

相反，他们在用眼睛阅读时，才能最好地记住有关信息。如果讲课的老师口才很好，那么前一类学生就更可能在课堂学习中获得成功；而在任课教师的可视辅助教学材料准备得很好时，更加偏好借助视觉手段进行学习的那些人将会如鱼得水。其实大多数人都很清楚，在学习时，自己到底是偏好来自听觉的讲解，还是来自视觉的讲解，但是却极少有人会根据自己的情况制定专门的进修计划。

如果你偏好借助听觉的学习方式，那么你就应该尽可能去聆听讲座，无论是现场的还是录音的，这些讲座都将使你有所提高。如果你能找到演讲的录音，那么就还有另外一项好处——你可以反复收听同一次演讲；而在现场演讲中，这是无法办到的。那些偏好视觉方式的求学者如果觉得有必要，可以反复地阅读一本书或者察看一张图表，这是他们的优势；但是实际上，有了录音记录（甚至于电视录像记录）技术之后，那些偏好听觉的求学者可以与前一类求学者具有同等效果的优势。

一位学习者的个体独立性，不仅会表现在其向书本或者演讲被动地获取知识的过程中，而且同时也表现在他应用这些手段时相应的行为上面。比如，有的人在聆听演讲时必须做笔记，否则学习的效果就很差——虽然此后他们可能根本不会再去翻看当时的笔记。实际上对于这部分人来说，在收听过程中的书写行为，会起到加强记忆力的作用。但是对另外的一些人来说，在听演讲的过程中作笔记，只会令他们更加分心；而只有把注意力集中于演讲本身，他们的记忆力才会达到最佳状态。有些任课教师会要求学生们在学期末上交各自的笔记，这类强制学生记笔记的做法，将会给后一部分学生的学习效果造成负面的影响。如果是自学，就不会有这类外来的压力；但是正因为以前在正规的学校环境中学习者曾经被强迫记过课堂笔记，或者通过老师的训练养成了这种习惯，所以有很多的人即使在自学时，也仍然继续记笔记。

人们的学习方式还有另外一种重要的划分方法：为了获得最好的学习效果，有的人需要动手实践（比如求解问题、编写程序等等）；而另一部分人则需要与其他人共同讨论。当然，这种差异并不是与生俱来、一成不变的，即使我们原来可能偏好于某一种方式，我们也可以从学习使用另一种方式的过程中受益匪浅。实际上，为了获得最佳的学习效果，我们的学

习方式也要因学习内容的不同而有所差异——当我们学习一种新的程序语言时，最佳的方式也许就是使用该语言来编程；而为了理解某个问题，也许最佳的方法就是在深入其中之前，先与他人作一全面的讨论。

关于这些特殊的学习方法，最后一个需要研究的方面就是人的工作习惯。有的人会在早晨起床后，对接触到的新东西接受效果最佳；而另外一些人则是所谓的“夜猫子”，整个上午的大好时光，他们的脑子都是昏昏沉沉的。有些人为了达到最佳的学习效果，需要端坐在硬质的平板靠背椅上，面前还要摆上一张书桌；但是另外一些人则是所谓的“享乐主义者”，他们在这样的环境中就会觉得不舒服，以至于连1加1等于2这样简单的内容都没有心思去学。有的人习惯于使用硬质铅笔，而有一些人则偏好软质铅笔。有些人会喜欢使用自来水笔、圆珠笔或者毡头墨水笔；而除此之外，另外一些人却离不开打字机——而且必须是某种特定型号的。有些人会喜欢带粗格线的纸，也有些人会喜欢带细格线的纸，而有的人则偏好不带格线的纸。此外，至少在某些类型的工作中，还有些人喜欢带有图案标尺的纸。在编写代码的时候，有一类程序员必须使用预先印好的代码稿纸，而另一类程序员则喜欢用普通纸。但还有第三类程序员，他们喜欢在打字机上或者终端上书写程序。即使是终端，程序员的偏好也不尽相同，有的人偏爱于使用图形终端，而另一些人则喜欢先打印出硬拷贝，然后再阅读。之所以有人会偏爱图形终端，其中一个原因就是，他们在嘈杂的环境中无法工作；但是我们也的确知道，另外一些人反而需要有一定音量的噪声相伴，才能进行学习——甚至于有的人习惯于一边学习，一边听收音机。

关于什么才是学习的最佳环境，人们争论不休。也许，这种争论注定不会有任何结论，因为就其本质而言，他们所争论的问题根本就不存在答案。如果正在创建一所学校，我们就很有必要采用一种就“通常的”学生而言是最佳折衷的运转模式。但是，如果是独自工作，那么又有什么必要舍弃最适宜于自己的学习环境，而非要去做那些折衷呢？即使在学校里，人们对效率的追求实际上也会经常超出合理需要的限度。比如说，为什么要强迫所有的学生使用相同型号的终端、相同规格的编码本或者相同规格的钢笔或铅笔呢？根据学校所能够提供的条件，采用一些更为合理的新方

案，就可以更好地鼓励学员们去发现最适于自己的学习习惯——只有这样，他们才能在告别学校之后保持不断学习的最佳状态，而不是觉得，要是从某一天起能够不用再去学习，那真是谢天谢地。

无论我们如何精心地为学习建立一个最佳的物理环境，如果不懂得如何利用所有可能的信息来帮助学习，再好的环境也不可能保证我们能够成功地完成学习任务。正如前面所讲的，我们可能会由于忽视了针对自身特质的需求，而没能在学习时获得最高效率；同样地，如果我们忽视的存在各种信息来源，或者没有注意到就在自己手边的信息，那么也会降低学习的效率。或许，造成这类学习效率降低的最主要原因，就在于那种认为事物总是非此即彼、非正即误的观念。虽然这种机械二分的思想不过是一个亚里士多德式的谬论，但是它却充斥于当今世界。这种错误观念的流行，使我们损失了难以计数的信息；而一旦我们能够挖掘出这些信息，就必将极大地提高我们在教育过程中的效率。

为了能够对信息的这种损失有所认识，请看这样一个例子。在一次测验中，有两名学生正在做一道算术题：25 - 16。如果其中一名学生给出的答案是“19”，而另一位学生给出的是“41”，那么他们都会被判定为“错误”。然而，尽管他们都没有能够算出正确的答案，但是这两名学生所暴露出来的问题却是截然不同的。要是他们所在的班级规模很大，那么他们通常都将被贴上“不懂得减法的学生”的标签。但是，只要其中任何一名学生运气好，能够遇到一位好老师的话，这位老师就可能从他们的错误答案中挖掘出更深层次的信息，并充分加以利用。

当我们需要自学时，更需要对所有这类信息加以重视。例如，要是我们正在编写一段程序，但是该程序就是不能“正常”运行，那么其实在程序的输出之中往往就会隐含着大量的信息。即使我们在收到程序结果时，确实有可能只有简单的一句“程序不正确”，但是无论如何，这种情况极为罕见。然而在调试程序时，似乎许多程序员都只对这句话有反应，而且他们的注意力只放在这里。反过来，只要程序的输出看起来完全“正确”，这些程序员就会立即把它仍到一边，再也不去关注。

学习是一个不断积极进取的过程。在学习程序开发时，程序员在两种情况下——无论是程序运行正确，还是运行错误——都需要积极地进行学

习。如果程序运行正确，你就需要把思路往回倒退 3 步，并且作一宏观的纵览，然后问问自己：我到底在哪些方面取得了成功？虽然这段程序运行正确，但是在编写其他看起来很类似的程序时，却遇到了很多麻烦？为什么我会遇到如此之多的问题？要是下次重新编写这段程序，我可以采取哪些不同的措施使工作变得简单？或者，如何使程序更为有效？或者，如何改进文档化工作？目前我可以从上述哪些方面着手开始？也许正是由于管理人员反复不断地催促尽快“完成”任务，所以一种经常出现的情况就是：一旦程序被认为是“可以工作的”，程序员马上就宣告其大功告成——其实这往往是一种误解。但是作为程序员，只要希望学习，那么你就必须一边顶住来自管理层的压力，一边花时间对自己的成功进行总结。而对于管理人员来说，一个好的做法就是：每当程序员宣称自己的项目已经“大功告成”时，给他一天的时间放松一下——这并不是一种奖赏，而是为了让程序员加深对该项目的理解。

其实，在程序无法正确运行时，我们会有更多的机会学到更多知识。即使程序员正在努力学习某项技术，但是只要他肩负巨大的产品开发压力，那么当他遇到问题时，就很自然地会想到如何通过一种他确定可行的“弥补方案”，来绕过这个问题——因为，这样他就不必去使用那些自己正在学习，但尚未掌握的新技术。比方说，假设有一位程序员正在学习 PL/I 语言中的数组表达式，而就在这时，他在正在开发的程序中遇到了一个数组表达式方面的问题。如果他因为认定数组表达式过于“复杂”，所以把它替换成自己从学会 FORTRAN 后就一直在使用的 DO 循环语句，那么他就将失去一次绝佳的学习机会。如果有朝一日你最强烈地感受到对学习的**需求**——比如就在你发现程序错误的那一刻——那么这个时候就是你进行学习的绝佳时机。

但是我们也不得不面对现实：在大多数环境中，与学习相比，产品开发工作更加重要；程序员可能会承诺说，只要他们掌握了新技术，就可以把现在用于学习的时间加倍地补偿回来，但是主管经理绝对不会容忍程序员因为要学习这些新技术而暂时延误进度。在面对其工作和学习之间的矛盾时，程序员应该如何处理呢？事实上，解决的办法异常简单。在经过适当次数的尝试之后，要是你还搞不清楚新技术失败的原因，而且没有任何



的进展，那么你就应该及时回头；如果你有某种“弥补的”方案，那么这时就应该采用。与此同时，你还应该构造出一个测试实例，以通过它找出原来所采用的方法之所以失败的原因。

在我们这个关于 PL/I 语言中数组赋值的问题中，该程序员可以在交付的产品软件中先使用 DO 循环，然后再着手去对数组的赋值进行单独测试。经过对测试样例的跟踪分析，你最后将明白问题所在。通过这个过程，你不仅可以学到此前不懂的问题，而且可以同时学到很多其他的东西——因为，与分析那些作为产品的软件相比，测试这类程序常常可以使我们学习得更好。

在这方面一个好的做法就是，在对软件产品实施“修补”**之前**，就应该把测试程序构造好。第一点理由是：一旦软件产品（哪怕是勉强）能够工作正常，人们总是很容易就会把这段程序抛到九霄云外——因此，我们必须强制自己执行纪律。但是或许更为重要的一点就是，有的时候仅仅通过构造测试实例，我们就有可能找出问题的症结所在。我们曾经见过这样一个例子，当时的程序员也在下面的这条 PL/I 的数组赋值语句上遇到了麻烦：

```
A = A / SUM (A);
```

由于他不清楚问题出在哪里，所以他就构造了一个单独测试 SUM 函数的例子，期望看到求出总和的输出。但是，就在他这样做的过程中，他忽然意识到：在上面有问题的那条语句中，SUM（A）不是单独计算的——事实上，每当对数组 A 中的某个元素进行赋值之后，该函数都会重新计算一次。通过对出错输出的分析，他也获得了同样的结论，于是他的假设就得到了证实。当然，由于这一次他不是像第一次那样，把输出结果的含义简单地理解为“错误”，所以他可以获得更丰富的信息。就这样，在最后完成的软件产品中，上述程序就被更正为如下形式：

```
B = SUM (A);        A = A / B;
```

在上述过程中，该程序员不仅学会了某些东西，而且透过他最终完成的产品程序，我们可以看到他对该语言新的理解。此外，他也经过了残酷的思想斗争，才战胜了自己对数组语句的畏惧心理，取得了最终的胜利。

在构造“学习的实例”时，程序员必须培养一种针对“临界测例”的感觉。除了有一点临界差别之外，这种测例与原始的例子基本相同。通过

对这两种测例的对比，程序员就可以准确地知道，应该分辨到何种地步才是最合适的——既不多也不少。比如，要是程序员在下面这条语句上遇到了问题：

```
INDEX ( X | | 'ABC','ABC' );
```

而他对 INDEX 内置函数不甚清楚，那么他就可以在其测试程序中构造几个极小的对比测例，每一个测例都与最初的情况在惟一的某个方面不同。就这样，也许有一个测例将使用一个变量，而不是表达式作为第一个参数，而另一个测例则可能使用一个长度不同的字符串；然后，第三个测例可能使用一个变量来作为第二个参数。每个测例都将为程序员提供一些信息，而这些信息互不相同。只要同时运行多个测例，他就很有可能获得自己想要的信息；而与单独的一次运行相比，这种可能性会大大提高。

为了实现这种积极主动的学习计划，程序员必须已经有充分的知识准备。例如，要是程序员对其程序语言掌握的不够纯熟，就不可能相对而言简单明了地构造出临界测例，也许会因此陷入种种琐碎问题之中难以脱身。因此，正规的教育计划必须明确这样一条目标：应该通过培训提高程序员的水平，使之能够应用其已经掌握的工具，不断进行学习。但是恰恰相反，我们在实际中常常见到的情况却是，正规的教育只知道一味地反复灌输——这种教育所起的作用，只会让程序员对自己所学的工具感到厌烦。在接过学位证书的时候，有多少毕业生会在心中暗自庆幸：谢天谢地，总算可以从书本和课程中间解脱出来了？如果一位劳动者讨厌自己所使用的工具，那么他就不可能干得很好；反过来讲，只有我们所使用的工具，才能够教会我们如何去成为一名出色的劳动者。

小结

为了提高程序员的工作绩效，我们可以从两个主要方面着手：培养他对工作的渴望，并使他们掌握为完成工作而必须具备的知识。前一方面称作“积极性”，而后一方面叫做“培训”（或者，如果其内容具有足够的一般性，也可以称之为“教育”）。但是，为什么程序员会更加卖力

地工作？或者从他们获得的利益来看，他们是否工作得太玩命了？——我们对这些问题所知甚少。而在对程序员的培训方面，尽管各类培训计划已经消耗了大量的资金，但是或许我们对此的了解反而更少。实际上，针对程序开发中的教育问题，有关的文章和论著已是连篇累牍，其数量之多，要远远超出程序开发心理学所有其他方面资料的总和。但是，这些资料大多是昙花一现，很快就变得没有意义。

也许，我们对正规的（学校课堂）培训所寄予的期望太大了。就本质而言，当前的教育体制依然在沿用中世纪的模式。既然如此，培训程序员的学校的情况也不会有什么两样。于是任何一个精心规划的教育计划，往往都不过是在按照业余人员的习惯，培训未来的专业人员——所以，也许他们还是不要这样去误人子弟的好。只要进入到实际的程序开发环境中去，你至少不会反过来忘记什么已经掌握了的技能。

能够将我们从计算技能培训的苦海之中解救出来的，正是计算机本身——虽然永远都是那样默默无闻，计算机却也会永远具有耐心；只要程序员具备学习的能力，计算机也可以永远地成为他们的老师。而在有教师相伴左右的时候，我们的学员却很可能还没等到学业完成，就已经对学习失去兴趣与动力。因此，或许应该为我们的学校制定一项目标，该目标就是要让学员们能够独立自主地去学习。也许这将代价不菲，也许可能不甚精致，而且也许还会显得过时；然而对程序员来说，计算机辅助指导（并非简单的 CAI[①]）仍是最佳的学习之道。

思考题

请主管回答：

1. 你认为，什么才是激励手下程序员的驱动力？对所有的程序员——无论男女老少——都是一样的吗？你是否认为，激励他们的驱动

① Computer－Aided Instruction，计算机辅助教学。——译者注

力与你自己的相同？

2. 你是否采取过什么措施，以防止手下的程序员工作过度，或者防止他们过分地投入到工作之中？你是否会嫉妒他们对工作的投入与敬业精神？

3. 对于为程序员受教育（不一定是前往程序员学校）提供条件，你的态度如何？为了鼓励程序员在工作中不断进行自学，你采取过什么措施？你之所以这样做，是否只是作为一个借口，实际上却是因为不愿意为他们去学校进修花钱？

4. 你是否鼓励程序员按照自己所偏好的方式工作？比如，你是否允许程序员在各自的办公室中安装不同的设备？是否允许他们把工作带回家去做？是否允许他们自行安排工作时段？

请程序员回答：

1. 为了保证你的工作又快又好，你的主管采取了哪些措施？你是否曾经与主管一起讨论过这些问题？如果没有，那么你是否认为他们了解这些情况？

2. 请写出你参加计算机程序开发培训的经历。并且标记出不同的经历都教会了你一些什么，哪些对你最有帮助。然后根据这个列表，请讲一讲自己在未来参加教育的计划。或者，你是否认为自己已经掌握了在程序开发时所需要的所有技能？

3. 你在接受新知识时最喜欢的方式是什么？在进修过程中，你如何利用自己所偏好的学习方式？

4. 你是否有某种“独特的”工作习惯，可以提高自己的工作绩效？你的主管是允许还是反对你按照这种方式工作？你的主管是否了解你的偏好？

参考资料

□ Sackman, Harold, *Man-Computer Problem Solving: Experimental Evaluation of Time-Sharing and Batch Processing*, Princeton, N. J., Auerbach, 1970.

再一次地，Sackman给我们讲述了许多有关程序员培训的重要问题。Sackman大多新近的研究，都是针对空军军官学校中的实习生进行的。在他的观察结果中，有一项证实了本章的观点——这就是：在帮助军校学员解决程序开发问题方面，截至目前为止最急需改进之处就在于“对教学方式的改进”。据我估计，问题的部分原因就在于严格要求工作必须独立完成——在军校中，这一点尤为强调。因此，尽管团队合作式学习方式在实际的程序开发工作中非常重要，但是学校还是认定这种方式属于“作弊行为”，并且予以废止。

请注意最后一章，Sackman在这一章中把他的发现与一般性的心理学文献联系了起来。他似乎认为，对于程序开发来说，经验与培训最终将成为决定程序开发成功的最主要因素。

□ Cofer, C. H., and M. H. Appley, *Motivation: Theory and Research*, New York, Wiley, 1964.

该书以专业的手笔，对有关积极性的形形色色的理论作了一个概括。正如心理学家们所认识到的，如果你是初次涉足关于积极性的研究领域，并准备对其经典著作有所了解，那么这本书堪称一本启蒙佳作。

□ Hilgard, E. R., and G. Bower, *Theories of Learning*, 3rd ed. New York, Appleton-Century-Crofts, 1966.

这本书成功地介绍了Cofer和Appley有关学习过程中积极性的心理学理论。同样地，这也是一本关于这个永恒主题的入门读物。

□ *Proceedings of the Computer Personnel Research Conference*, Association for Computing Machinery, New York, published annually.

关于程序员培训的方案和体制的文章，零星分布于计算领域的各类文献之中——就像布丁中的水果一样。而在该论文集中讨论培训的那些文章，则可以为读者提供足够的信息，并进而使他们获得一个整体印象。

□ Hall, Douglas T., and Edward E. Lawler III, Job Pressures and Research Performance, *American Scientist*, Vol. 59 (Jan.-Feb. 1971).

尽管我们尚未就工作压力对程序开发工作绩效的影响作一研究，程序开发的主管经理还是可以从这篇文章中获得启发。当然，为了能够有所收获，他们的头脑只需稍

微开一点窍。正如我们已经知道的，他们必须认识到：程序开发的行为与所谓的“研究”行为在很多方面不尽相同。这篇文章指出，至少对于那些复杂的智力活动来说，各种各样的工作压力与产出率之间的关系并不简单；而要是将该问题过分简单化，主管经理就将自食其果。

第10章　积极性、培训以及经验

评　　注

在深入学习了MBTI人格理论之后，我对积极性的理解要比25年之前深刻了许多。比如我现在所认识到的，像我这种人格类型（INFP）的人很容易被那种希望帮助人们自学的想法——这也是我撰写本书的目的——而打动。属于其他一些MBTI人格类型的人，也同样可能受到这种想法的激励而提起笔来，但是其目的不尽相同。比如INTJ类型的人，他们可能会希望向人们解释，如何剥去程序开发的神秘外衣，以发现其中若干本质的、优美的模式。

显然，MBTI模型并不能解释所有各具特点的积极性——尽管这种尝试的成功机会并不小。为了获得一张参加迈克尔·杰克逊演唱会的门票，有些人什么事都可以做；但是对我来说，我更希望这是一张迈克尔·乔丹比赛的门票。虽然任何理论都无法解释我们各自的嗜好，但是机灵的主管却能够轻松自如地了解不同人各自不同的积极性。他们的方法很简单——答案就在自己的鼻子下面！当然，这种方法要求在主管与雇员之间的建立某种关系；但是在某些类型的管理体制下，这种关系是不存在的。在这种情况下，主管经理们只好努力去猜测手下人的工作积极性到底是什么。

在过去的几十年之间，程序员身上所发生的变化之大令人吃惊，其中之一就是他们在校期间接受培训的多样性减少了。时至今日，在一支系统程序开发团队中，虽然成员之间的差别更大了，但是他们的经历却大同小异，其中绝大部分成员都受过“计算机科学”方面的正规培训。

例如，我们去年有一次曾为一家大型软件公司做咨询，其间我与一支由 5 位开发人员（尽管他们还不是系统程序员）组成的团队一道工作，而这 5 人全部都拥有计算机科学专业的学位。不过，只有两人是直接通过学校获得学位的。在前来这家公司工作之前，有两个人曾在各自的工作中大获成功：其中一位还在业余兼职，他拥有自己的蓝草音乐[①]乐队；而另外一位在晚上还要去上法律学校，并希望有朝一日能在家庭法律方面一显身手。而第 5 位程序员虽然早在 12 岁时就已经通过自学掌握了程序开发的技能，但是在为该公司工作了 5 年之后，最终还是被送往学校进修，并接受了正规的教育。

然而这些人最为有趣的巧合是，他们都是在到学校去学习程序开发技能**之前**，就已经懂得如何进行程序开发了。这种巨大的变化，是在个人电脑的出现之后才得以发生的。而在 1956 年来到 IBM 工作之前，我根本都没有**见过**计算机是何等模样。就在此之前，当我还是一名学生时，就曾经为物理系工作过，当时我的作用居然就是作为一台“**计算机**”——手工计算 10 乘 10 矩阵的逆矩阵，所以当时我的工作头衔就是“计算机”。

我的后代在学校期间就接触到了计算机：我的儿子 Keats 就获得了计算机科学专业的学位；而早在**进入**学校之前，我的孙子 Nicolas 就已经胜过了他那位极具耐心的“老师”。虽然他的程序开发心理或许会与我们的有所不同，但还是让我们静候本书的金版[②]吧，那一天才是做出最后结论的时候。

① Bluegrass，蓝草音乐。一种起源于美国南方的民间音乐，通常用班卓琴或吉他等非电声乐器演奏，特点为节奏轻快并像爵士乐那样即兴演奏。——译者注

② Golden Edition，50 年后的再版。对本书而言，也就是 1971 + 50 = 2021 年。——译者注

Chinker

第Ⅳ篇

程序开发心理学

Computer Programming Psychology

第 11 章　程序语言
第 12 章　程序语言设计的一些原则
第 13 章　其他的程序开发工具



程序开发工具

……我的听众们，我有一个建议给你们，这个建议也适用于教授们。鉴于每个人都知道，只要我愿意，我完全可以使用数理逻辑，所以我被允许使用通俗易懂的英语。请考虑下列命题："有的人在妻子亡故后，娶了她的姐妹"。我能够用一种需要经过几年的考证才理出头绪的言语来表述它，而这会使得我更加自由。我向年轻的教授们建议，在写他们的第一篇著作时，要使用只有少数博学之士才能理解的深奥的行话。经过这样的铺垫，从此以后每当他们必须要说些什么的时候，总可以使用一种"所有人都听得懂的"的语言。如今，我们的命运被牢牢地掌握在教授们的手中，我不得不认为：如果他们能采纳我的意见，他们应该受到我们的尊敬。

——伯特兰·罗素 Berfrand Russell[①]

① 摘自伯特兰·罗素的"Portraits from Memory"，1951 年、1952 年、1953 年及 1956 年版权。经过 Simon 和 Schuster 和 George Allen 和 Unwin 有限公司的允许，再次印刷。

如果说笨拙的工人会厌恶他的工具，那么技术精湛的工人就会讨厌简陋的工具。在某种意义上，劳动者的工作任务，是由他所使用的工具决定的——工具经常被用来作为其使用者的象征，让我们看看一些例子：木匠使用的曲尺、泥瓦匠使用的铲子、勘探员使用的经纬仪、摄影师使用的照相机、工人使用的锤子、农民使用的镰刀。即使是手艺最精巧的工匠，如果使用的是有缺陷的或者设计不佳的工具，也只能制造出低劣的作品，同时也会因此沦落为一个技艺拙劣的工匠[①]。任何一名工匠，只要他确实希望在其职业生涯中创作出最好的作品，那么就不会接受这类工具；任何一位雇主，只要他欣赏的是作品的品质，也不会要求其工匠去接受这类工具。

在程序设计中，情况是否不一样呢？乍看起来，似乎确实有区别，但是引起这种误解的根源在于，与其他手工艺相比，程序设计还显得很不成熟。任何行当使用的工具都经过了长时间的演化，但是程序设计在发展到今天这个水平之前，只经历了短短的一、二十年。正因为如此，程序设计的工具缺乏统一的标准，在使用现存的最好工具方面，没有任何普遍适用的经验可以借鉴。

即使是最粗糙的汇编语言，也具有很多优点，而正在使用机器语言的程序员，也许根本没有意识到这些优点——而使用机器语言的情况，要远比想像的普遍。在我们的程序开发心理学课程中，曾经有一名学生做过一次调查，他对其所在的程序开发工作室中出现的 2000 个程序开发错误进行了分析；这个工作室里的所有程序员都使用机器语言，为小型的专用计算机编写代码。这些错误中，有 65%的原因可以归结为各种细微而琐碎的问题，比如：为实现延迟而插入的空操作指令数目不对；操作码不对；为了实现间接寻址等特性，错动了一个比特位；使用错误的绝对地址；或者是位移量计算有误。而只要使用设计最为简单的汇编语言，这些程序员就可以轻易地消除其代码中错误的 2/3——但是，他们竟然不知道这种可能性。

对程序开发的帮助可能来自不同层次，当我们考虑更高层次的工具

① 这使我想起一句中国的谚语：工欲善其事，必先利其器。——译者注

时，我们发现，上述情况不断地重复出现：使用汇编语言的程序员，可能会对那些使用机器语言的人嗤之以鼻；但是对于那些使用支持宏机制的语言的用户而言，他们却又成为消遣的笑柄。但是，并不是所有这些人的沾沾自喜，其原因都可以归结为一种系统相对另一种系统的优越性。人们总是有一种不安全的感觉，担心自己的程序开发系统，并非处于人类技术成就的巅峰；确实地，上述情况中的相当一部分，不过是有意或无意地隐藏这种不安全感的一种方式。

在本书的这个部分，我们将力图从心理学的角度，对程序员使用的主要工具进行考察；针对目前围绕系统设计与选取所进行的无知的争论与争吵，我们期望通过这些考察，确认这方面的争吵与争论是否真的无法消除。毫无疑问，这样高的目标不可能实现，但是我们依然希望，至少能够为此方面的研究开个好头，因为与此前研究过的一些方向相比，这个方向给人们带来更大的希望。

第Ⅳ篇　程序开发工具

评　　注

每当我这老朽昏花的眼光，落在今天的程序开发工具上时，我真是有些垂涎欲滴——但愿不只是因为衰老。啊！想像一下，如果当年有这些工具的话，我们应该可以完成什么样的工作……！好吧，实际上，我们也许还是会和以前做的一样——制造更多的工具——这也是任何真正的专业人员会做的。

在我早年的大部分职业经历中，我都是在开发各类工具——操作系统；调试工具；性能分析程序；静态和动态代码分析程序；测试样例生成器、连接器以及模拟器；数据库系统；解释器等诸如此类。你也许注意到漏掉了编译器。由于某些原因，我对编译器的开发工作从未产生过兴趣。实际上，从事编译器开发的人一直都很多。时至今日，这种情况也没有变化，开发编译器的人数依然很多。

但是有一件事我很感兴趣，这就是为编译器配备元语言以及其他工具，因此我曾经一度偏离主题，进入到 MetaCOBOL 的领域。与 Applied Data Research 公司的伙伴们一道，我还写了“*High-Level COBOL Programming*”，这是一本有关 MetaCOBOL 的书。我们在这本书中作了一次尝试，通过一种宏语言的形式，将 COBOL 转换为其他更好的语言。这是我为语言提供工具所作的最后一次主要尝试，总的来说，这次尝试没有成功。高级的工具只能延长低级语言的寿命。

有好几年，我都在为报业辛迪加的一个面向专业程序员的专栏撰写文章，目的在于促进更优秀工具的产生，以及对现有工具更好的利用。

一部分这些专栏文章被收集到“*Understanding the Professional Programmer*”和“*Rethinking Systems Analysis and Design*”之中。这些书籍和专栏文章对促进优秀工具的开发到底有多少作用，我表示怀疑——这当然不能与个人电脑产生的影响相提并论。

远在20世纪50年代至60年代初期，程序开发所使用的工具，大多是由用户自己来设计和制作的。比如在开发我们的第一个操作系统时，我们是与IBM的众多客户联合完成的。IBM（我当时作为IBM的员工一道）参与了该项工作，但只是合作者中的一员。因此在那个年代里，我们的所有工具，都是通过SHARE协会[①]或者与IBM制造的硬件相捆绑的形式，完全免费发行的。

到60年代后期，以ADR的Marty Goetz（他曾与我一道为MetaCOBOL并肩努力）为首，从法律的角度成功地对捆绑发行软件的观念提出了批判。这些批判的结果是，软件可以独立进行销售（或出租），但是刚开始时，硬件制造商们仍然在软件工具方面拥有很大的控制权。

到80年代，个人电脑的出现导致了软件工具开发的民主化，于是整个局面再次大为改观，使用工具的用户们又从新拥有了对工具的决定权，同时使得各种工具之间的竞争成为可能。这种竞争不仅扩大了对软件工具的需求，而且降低了其价格。这就是为什么对于像我这等老朽们来说，当今的软件工具简直像是乌托邦中的传说一样。

① SHARE：Society of Help to Avoid Redundant Efforts，IBM公司的计算机用户协会，目的在于协助避免重复工作。——译者注

第11章 程序语言

讨论程序语言的文章数不胜数，但是其中关于心理学的却寥若星辰。有一位作者如此表达关于程序语言最典型的观点，那就是“真正地尝试去理解程序开发及程序语言的完整含义”。虽然他的观点值得尊重，但是对于他所诠释的“完整含义”，即程序语言与数学基础，以及与现实的及设想的计算设备之间，是什么样的关系，我们却不敢苟同。这种观点丝毫没有提及人的因素，但那些万不得已才使用程序语言的可怜的家伙们（那些花了点时间凭空想像计算设备的人或许应该除外），怎么能算得上程序语言的典型用户呢？

本章我们也会尝试去讨论程序语言的完整含义，但首先我们考虑的是“人”，其次才是机器和数学，然后再结合心理学进行讨论。在本书的前些章节，我们花费了大量的时间和篇幅讨论什么是“程序开发”。这一部分，我们对程序语言的研究，也将首先讨论“语言”是什么，以及程序语言与自然语言的关系。在此基础上，我们会花些精力讨论程序语言设计。鉴于多数程序员对语言设计兴趣不大，我们也将提供一些从我们已经设计好的程序语言中进行选择原则，以满足广大程序员在这方面的需求。

程序语言与自然语言

如果说有哪个领域每个人都是专家的话，那就是语言。每个人都会说，有些人还会读写。如果一个外国人这么说英语“I would like of

dinner”，每个美国人知道他用错了介词，包括那些根本就不知道什么是介词的人。一般的人对语言本身的理解并不深刻，他们很可能把某种仅仅是表面上看起来像语言的东西叫做“语言”。因此，人机交互所使用的那套符号系统被称为“程序语言”就毫不奇怪了。事实上，历史上第一位程序员，Lovelace 夫人，好像早在 1846 年就认为程序语言是一种“语言”了。她虽然身为著名诗人拜伦的侄女[1]，但其认为符号系统亦可称为语言的想法却并非源自她的文学知识，而是数学知识，因为似乎在很早以前，数学就被称作是一种语言了。因此，“程序语言”的想法实际上是与“编程序”同出一辙。

虽然“程序语言”并非真正意义上的语言，但鉴于这个概念由来已久，没有必要在“是”与“不是”这个问题上纠缠了。语言是什么呢？说它是什么，它就是什么。所以，把人与计算机交互的通讯系统囊括在人与人或动物与动物之间的交流体系之中，冠之以“语言”，并无任何不妥。反过来，研究一下程序语言与其他语言存在的差异倒是可能的。

Hockett 曾经列出了人类自然语言（言语）的 13 项结构特征，可以作为比较程序语言与自然语言的出发点。在这 13 项特征中，首当其冲的是“听说信道”的使用——也就是说，我说你听。这个信道的优点不一而足，尤其是解放了手和其他器官，他们可以去忙自己的事情。通常程序开发是一项手的运动——如果有人请你“说点 FORTRAN”，你的反应是什么？好笑！因为 FORTRAN 是一种书面语言，它固然拥有一些书面语言的特征，但同时它又与其他书面语言区分开来——不存在与之对应的口语体系。这里并不是说书面语言仅仅是口语的书写形式；而是说书面语言，比如英语，深受口语的影响。对于其他书面语言，比如基于汉字书写的语言，这种相互的影响虽然没那么强烈，但总是存在的。

在自然语言中，口语是新语汇产生的重要源泉；程序语言缺少口语形式，导致的一个后果就是缺少创新，发展缓慢。另一个后果是，如果没有黑板或者纸笔，讨论程序语言会很困难。因此每个与编程序有关的办公室都应配备黑板、粉笔和许多黑板擦。

[1] 实际上应该是拜伦的女儿。——译者注

Hockett 所列的第 2 个特征是“迅速消失”。我们之所以需要许多黑板擦，是因为在讨论程序时，黑板上所写的东西不会像说的话那样自动消失。程序语言如果也具有“自动消失”功能，有时也方便不少。在百慕大群岛安装计算机时，我们曾养成了一个习惯，把程序开发的粗略想法写在海浪拍打的珊瑚沙之间。大海注意到了每个想法只在值得保留的期限内存在，过期之后就被海浪无情的冲走了；而在平常编程序时，我们必须事前做好准备，删除过时的东西，除非我们有一台图形终端。

第 3 个特征是“广播传输和定向接收”，也就是说，说话时我们或多或少是同时对着各个方向的，而在听的时候，我们能够判断声音究竟是从哪个方向传来的。程序却恰恰相反。程序与祈祷类似，单项传输并广播接收。天堂接收祈祷，计算机接收程序，不分来源，一起照单全收。那么，我们也许会想，当信息从计算机出来时，程序语言允许它有什么样的变化。不管怎样，反过来肯定是对的。无论听众是谁，“系统”只管对他们广播同样的信息。这种机制存在着一定被修改的可能性。

也许（当今）程序语言与自然语言的真正重要的差别在于 Hockett 的第 4 个结构特征——可互换性。尽管我们经常会碰到听不懂别人说话或者说的话不被人理解的时候，人类语音交互的一个普遍现象是每个人都能把他所能够理解的话在脑海中重现。简言之，就是我们“说的是同一种语言”。在跟计算机交流时，很不幸，通常我们说的是不同的语言。无论程序语言有多少毛病，他们依然占上风。至少“死机”时乱七八糟的局面跟我们放进那该死的机器中的东西差不多。我们放进去的是 FORTRAN 或 COBOL 的程序时，出来的结果却往往不可思议。有句话说得好，每当计算机有重要的结果要告诉我们时，它就开始胡说八道。

好在现在有一种趋势，要通过程序语言在人机对话中引入可互换性。而且，这样做势必对程序开发的效力产生重要的心理方面的影响，尽管它可能会降低机器的效率。但是，另一种不可互换性情况更严重，而且跟机器的关系更小。程序不仅用在人与计算机的交流中，同样也用在人与人的交流中。同样两个人，当他们用程序语言交流时，似乎不像我们想像的那样，跟自然语言一样具有可互换性。这个问题的极端情形是，一个软件开发主管能读懂 COBOL 程序，但是不会写，或许也不能跟程序员讨论

COBOL，那他们的交流就成问题了。

在 Hockett 提到的其他结构特征方面，程序语言的确能赶得上甚至超过了自然语言，比如构词能力，也就是表达以前从未有过的事物的能力。在某种程度上，程序语言的设计反映的使用模式与自然语言中常见的有所不同。对自然语言而言，上面所说的构词能力固然重要，但对程序语言来说，表达新事物的能力是不可或缺的。

程序语言与自然语言最本质的区别在于语言的功能，而非实现其功能的结构特征。Bruner 给出了自然语言的 6 种功能——感叹、命令、陈述、元语言、诗歌以及交际。尽管在程序语言中，我们可以一一找到他们的影子，但其重要性和使用的频率与自然语言相比都有很大差别。

感叹句包括“你好!”以及“胡说八道!”这样的句子，是说话人感情状态的一种形式化的表达。通常而言，计算机对程序员的感情状态不感兴趣，尽管它的形迹可能会潜移默化地影响到程序。交际性语言包括诸如“呃……唔……但是……而且……”，其作用是保持交流渠道的畅通或测试其运行的状态。在某些终端系统中，用户在思考时不停地敲击“上档”键，以避免他的程序被系统踢到优先级队列的栈底去。这个动作可以看成是自然语言中的语气词“而且”的翻版。如果不敲着“上档”键，计算机就会长时间与用户失去联系，从而假定降低他的程序的“活性”是没有问题的。当然，叮叮当当地敲击“上档”键破坏了系统调控作业进度的能力，但更重要的是它可能会破坏程序员的思考过程。但愿这种无意义的通讯能够从程序语言业已混乱的领域消失。

乍一看诗歌语言在程序开发中根本没有立足之地，但它所扮演的角色应该日益受到关注。在诗歌中，从审美的角度看，语言本身的模式是最重要的。但在编程序时，有没有像 Keats 这样优美的诗句呢?

O Attic shape! Fair attitude! with brede
Of marble men and maidens overwrought,
With forest branches and the trodden weed;
Thou, silent form, dost tease us out of thought...

如果我们指的是大声朗读出来，答案是“绝对没有”，因为我们已经了解程序语言不是口语，更不用说诗歌了。但是，纯粹从形式上来看，难

道真的没有“令我们怦然心动的”东西吗？

程序员时常会把自己的程序当作一件艺术品，追求美感。若非如此，他就不能称之为一个真正的程序员。这一段不对称；那一段太笨拙，不流畅；整个程序的格局看起来不美观。在程序员中间，粗糙和实用主义固然流行，但内心深处，每个人都知道，对程序来说，仅仅能工作是不够的，在别的方面也应该“对头”才是。后面在讨论程序语言设计和程序测试时，我们将会发现，程序的审美和实用两方面的价值并不是对立的——越美的程序越可能是对的。或者，更诗意一点，“美即真，真即美”。

程序的写作充满了陈述句和命令句。陈述句是对事实的描述或者提问，比如“那是海蒂”，“这是一个定点二进制数，由13位有效位和一位符号位组成”。命令句发出命令，如“别拿我开玩笑了”，“求出这个矩阵的逆矩阵”。我们以后会对这类语言详加阐述。不过，这里我们应该注意到，尽管在自然语言和程序语言中，这两种句型都很常见，但后者的论域（Universe of Discourse）范围却远远小于前者。因此，尽管在PL/I中，对赋值语句或声明的数量并没有任何限制，但是跟英语相比，也只是九牛一毛。我们已经对程序语言中论域大小的重要性有所了解，但正如我们希望说明的那样，其对程序语言心理学的重要性更大。

元语言语句是用来说明语言的，他们对程序语言而言是最重要的，尤其在学习阶段，编写编译器之类的程序以及设计程序语言的时候。程序语言之所以不同于自然语言，在于他们单纯地依赖自然语言作为元语言，可能还会需要一点点数学。这种情况一直延续到最近。这一章其实就是用元语言描述程序语言的一个例子，用的全是英语。不过，现在人们越来越多地把注意力转移到了一些更擅长做元语言表达和推理的形式语言或非形式语言上来了。而且，有些程序语言已经开始集成强大的元语言功能——使得我们不仅能够处理程序语言，而且能够处理关于程序语言的断言。尽管这些努力的前景如何尚未可知，也没有历史经验可以借鉴，我们仍然希望能就元语言功能在程序语言心理学方面能发挥的作用稍作分析。

程序语言设计

谈到程序语言设计的心理学原则，就不能不讲“天才裁缝 Levine”的故事。这个故事说的是有个人去找 Levine，请他给自己做一件便宜的衣服。衣服做好了，他穿上一试才发现根本不合身。“你瞧，”他说，“这件衣服的后襟太长了。”

“没事儿，”Levine 一边说，一边教他如何弯下腰拱起背把后襟抬起来。

“可是，那右胳膊怎么办？袖子太长了，长了 3 英寸呢。”

“没事儿，”Levine 又说，教他歪着身子，右胳膊使劲往外伸，这样袖子也算合适了。

“那裤子呢？左腿太短了。”

“没事儿，”Levine 还这么说，教他把臀部抬起来，往上拉左腿。虽然他一瘸一拐地，但衣服看上去还凑合。

这个人再也没有什么可说的了，心想被 Levine 骗了，只好窝着一肚子火悻悻地走了。还没走多远，就碰上一个陌生人拦住他问，“请问，你穿的是件新衣服吗？”

这个人心情略微好了一些，竟然有人注意到他的新衣服了，因此他并没有任何不快。“是啊，”他回答说，“你问这个干吗？”

“呃，我正想给自己做件新衣服呢。你的裁缝是谁呀？”

“是 Levine，喏，就在那条街上。”

“多谢了，”陌生人赶紧往那儿走。“我决定去找 Levine 做衣服了。为什么呢，能给你这样的瘸子做这么合身的一身衣裳，他肯定是个天才的裁缝。”

把这个故事改编一下，变成“天才语言设计师 Levine”如何？讨论语言设计所面临的第一个难题是我们不知道这个问题的答案。我们不知道语言设计者们是不是天才，或者我们这些普通的程序员是不是瘸子。一般而言，在终于克服了心理障碍，学习了一门新语言之后，我们才发现以前使用的语言是多么土。换句话说，我们的标准是变化的——在程序语言设计

中要充分考虑到这一点。

为什么在这件事上我们的标准这么松而在别处却那么严呢？问题在于机器的严密性，进而导致程序语言的严密性。每当一个人面对1台新机器，他必须要做出选择，要么调整他自己，要么调整机器，以减小现实和理想的差距。尽管机器，尤其是计算机，是可以调整的，但所需要的时间却远远大于人。程序语言也想调整计算机，让他们更符合人类的习惯和局限性，但同样遭受更新周期过长的困扰。因此，用户就面临着两难的选择，是像许多人所做的那样，等上几个月甚至几年的时间，开发自己的程序语言呢，还是抛开牢骚，磨平棱角，学会在现有东西的基础上生存。

人文主义者经常说机器会剥夺人的人性，让他们变得冷酷无情，然而实际上，反过来也是对的。因为机器是严密的，所以使用机器的人们如果想要成功，就必须变得更宽容。也许这就是人文主义者所说的“非人性化”，因为，在人与人的交流中，双方的付出和索取是平等的。一方完全给予或完全索取的关系是不符合人性的，而且会导致一方或另一方的人格扭曲。

在适应某种程序语言的过程中，我们很容易养成对它的依赖性，只是因为我们对其付出得太多了。我们经常听到男同事抱怨他的老婆多么唠叨、懒散、花钱如流水。但当你问他为什么不离婚时，他说没有她他活不下去。即便维持现状很痛苦，大多数人宁愿忍受，也不愿意放弃一个熟悉的伴侣，尝试未知的新事物。当我们试图教一个程序员另一门程序语言时，可以清楚地看到这一点。教他第1门语言是没有任何问题的，因为他对哪门语言都没有投资。等他学会了第2门或更多，他就会知道这个世界上他不懂的事还很多。可是，让他放弃第一门语言就好像告诉他肯定会有痛苦却不一定有快乐来补偿一样。

如果我们能够阐述并教给他们一些跟语言无关的原则，使得初学者也能掌握一些与语言关联性小一点的度量方法来评判他正在学习的语言，也许情况会有所改善。但目前大学和软件学院的教学实践似乎恰恰走反了方向。学校非但不传授通用的原则，或者至少努力同时对照着教两种语言，或者，最低限度地，教一种具有广泛代表性的程序语言，而是致力于教学生用单一的、简单的、人工的语言编程序。目标似乎是让学生能尽快写出程序——并

不是说这样做不值得——但代价却是限制了程序员的未来发展。

公平地讲，我们应该区别对待专业和业余的程序员。学校似乎热衷于制造大批的业余程序员——可能他们认为专业程序员能够而且应该自己照顾自己。可是，当语言设计者开始认为给业余程序员和给专业程序员设计语言可以用同一套指导原则时，事情就麻烦了。

为了将程序语言设计背后的那些原则，与心理上的现实相吻合，我们必须开始认真考虑基本的经验数据。读程序会有所帮助，但光读是不够的。读程序时，我们很少能肯定地说出为什么需要某段代码。要理解程序语言的心理学特征，除了读程序，我们还需要做更多的事情——必须观察编程序的过程。即使这个过程——即使采访程序员——也并不能得到我们想知道的一切，而且我们还必须凭借大量的资源和设备，才能得到比直觉稍微强一点点的结果。为什么要搞这么复杂呢？其中一个原因就是程序员自己也经常弄不明白他在做什么以及为什么做——在研究人类的任何行为时，这都是个普遍存在的问题。

举例来说，一个程序员在编 FORTRAN 的程序，他想使用（J－21）这样的下标表达式。他知道有一些明确的、没有歧义的规则规定了什么样的下标表达式是合法的，什么样的是不合法的。但在这种情况下行不行，他不是很有把握，那他该怎么办呢？当然，他可以去查手册，但如果他是个聪明人的话，他就会事先把下标值算好，用

K = J－21

K **肯定**是可以做下标的，而且这样做比查手册快多了。固然，程序的效率可能会低一点，但此时此刻，这不是他脑子里考虑的最重要的问题。

那么，为什么这个程序员要这么做呢？是因为 FORTRAN 语言的局限性？可是，你会说，J－21 这样的下标 FORTRAN 是认的，那肯定是程序员自己的能力有限——他对 FORTRAN 的理解有限。虽然这种论辩找到了过失方，但在设计或评价程序语言时，它会误导我们。因为，在 PL/I 中，程序员很可能就不会犯这种错误。为什么？因为他很可能知道，PL/I 对下标没有任何限制，因此他根本无需担心一个表达式是否合法。

当我们从心理学的角度去评价一种程序语言或者一个机器时，不能奢望把拙劣程序的责任都推到可怜的程序员身上。如果同样的程序员使用 A

语言总比B语言好，一个很好的论点就是，只要他们足够聪明能掌握B语言，他们也能做得更好。当然，这中间牵涉到时间的因素，因为人们需要时间去学习使用程序语言或机器。因此，对这些东西的评价依赖于整个系统，包括课程、教材、语言、锁编译器提示信息、他人的帮助以及其他任何能导致一种语言比另一种用得好的因素。在一个系统中，终端上显示的错误信息可能要比另一个系统更有启发性。如果被选来用机器语言而不是FORTRAN编程，就可能会让程序员觉得自己更重要。如果我们希望有效地提高语言的质量，就不能把这些因素置之度外。

我们很难把程序开发的效率低下归咎于程序员或者程序语言中的任一个。在某种程度上，原因在于如果我们拥有理想的程序员、程序语言就是多余的了。妨碍我们直接使用机器语言书写问题说明的，是**心理上**的障碍。让我们来勇敢地面对这个问题：人和计算机的思维方式不同——这就是我们为什么要用计算机。程序最多不过是两个不同物种之间的交流，而程序语言以及系统设备不过是让这种交流变得简单些的工具，对其中一个物种而言。哪一个呢？自然不是计算机，因为从来没有人听计算机抱怨过，说它做不来。

小结

在“程序语言”一词中使用“语言”这个限定词，在诸多方面妨碍了人机交互的发展。称之为语言并不能使其成为真正的语言，反而会导致错误的类比，从而误导研究方向。如果希望在程序语言方面取得进展，我们就必须放弃企图用“真正”的语言来编程序的浪漫想法，因为程序语言永远不可能和人类的自然语言一样。我们可以要求程序语言更“自然”一些，但绝不是要他们跟英语或其他在建筑通天塔的过程中使用的语言完全相同，而是说表达的方式和表达者的思想要协调一致。但最终，为了更好的“匹配”，用户可能不得不受点委屈——为了跟计算机合拍，或许最终我们不得不改变自己的想法。为什么不呢？人类的每个发明创造不都改变了人类自身吗？

思考题

请主管回答：

1. 黑板是不是你的程序开发环境中标准配置的设备？你是否曾经拒绝过申请黑板的要求？

2. 你的公司是如何选择该用哪种程序语言的？

3. 叙述一项你的公司更换程序语言的经历。

请程序员回答：

1. 你所使用的语言的设计者是天才吗？从你的亲身经历，举例说明他的创造性。

2. 叙述你学习第一、第二、第三门程序语言的经历，请尽量从你学习的顺序，而不是从语言本身出发，指出学习过程中心理上的差异。

3. 如果你是个资深的 FORTRAN 程序员，你能否给出下标表达式的准确规则？你有把握吗？当你对一条规则没有把握时，它会对你编程序产生什么样的影响？你会去查手册么？

4. 程序员的身份有没有改变你在不编程序时的思考方式？如果是，试举例说明。

参考资料

□ Morrison, Phillip, and Emily, eds., *Charles Babbage and His Calculating Engines*, New York, Dover Publications, 1961.

除了 Babbage 关于他的计算机器之外，我们还可以从本书中找到 Lovelace 夫人的评注，她的这些评注清晰地显示出，她对程序语言的许多基本原理已经有所理解。如果有谁天真地认为知识的发展总是单调递增的，那么他就应该仔细地读读这本书。读完之后他们应该问问自己：在 1860 年到 1940 年之间，关于程序语言基本原理的理解经历了什么样的

发展历程？当时的 Babbage 尚不能从原理上，将他的计算机器与其他的机械装置清晰地区分开来。或许正是因为这一点，所以他的工作才会凌乱地分布在世界各地的图书馆，没有引起人们的重视；直到他的发明后来又以电和电子装置的形式重新被发明出来。在关于程序语言的这个问题上，我们是否有什么教训可以吸取呢？

□ Zemanek, H., Semiotics and Programming Languages, *Communications of the ACM*, 9, No. 3 (March 1966), pp. 139－43.

Zemanek 曾经担任过设在维也纳的 IBM Vienna Laboratory 的主任，当时该实验室正在为 PL/1 语言开发 Universal Language Document (ULD)。它也许是有史以来人们所尝试完成的最为野心勃勃的一份元计算文档，并且也可能是最成功的一份。Zemanek 在这篇论文中介绍了其构思的一些结构，这些构思显然对 ULD 的发展过程产生了影响。

□ Greenberg, Joseph H., *Universals of Language*, 2nd ed., Cambridge, Mass., M. I. T. Press, 1966.

本书是在 1961 年的一次会议的基础上完成的，但是作者并没有逐字逐句的生搬硬套。如果要了解关于各种语言的普遍性特征方面的研究，那么本书就是一份很好的导言。像 Greenberg 这样的语言学家当然不会认为程序语言也能被称为语言；虽然如此，对于程序语言的设计师而言，除了 Hockett 的论文之外，本书中所包含的其他至少 6 篇论文都会很有启发。

□ Bruner, Jerome, *Toward a Theory of Instruction*, Cambridge, Mass., Belknap Press of Harvard University, 1966.

Bruner 在本书中列举了语言的 6 项一般性功能，以及他的很多其他思想。在有关语言、学习和思考等的研究领域中，Bruner 被认为是最重要的现代思想家和研究者之一；本书也是了解其思想的很好渠道。

□ Miller, George A., *Language and Communication*, revised ed., New York, McGraw-Hill, 1963.

程序语言设计师只有在读过本书之后，才能获准继续去设计另一种语言。正如 Miller 在他的导论中所讲的："对于语言的科学研究方法与投机取巧式的清谈恰恰相反，这种方法是从直接对正在交流的个体进行观察入手的，然后从大量的观察结果中找出其相互关系，最终得出符合实际的科学知识。"

□ Vygotsky, Lev Semenovich, *Thought and Language*, Cambridge, Mass., M. I. T. Press, 1962.

Vygotsky[①] 是一位伟大的心理学家，可惜他英年早逝。但是他关于思维与语言的这本

① Lev Semonovitch Vygotsky，维歌斯基，20 世纪初俄国心理学家及符号学家。生于 1896 年 11 月 5 日，至 1934 年 6 月 11 日去世时尚不足 38 岁。——译者注

译著，时至今日仍然堪称此方面最具启迪作用、最值得一读的书。只要愿意一读此书，每一位程序员都会受益匪浅。

□ Steel, T. B., Jr., ed., *Formal Languages Description Language for Computer Programming*, Amsterdam, North-Holland Publishing Co., 1966.

这些论文都来自一次由 IFIP① 组织的，关于形式语言描述语言的国际会议。所谓的形式语言的描述语言，也就是程序语言的形式元语言。这些论文的水平虽然参差不齐，但是对于那些对语言的语言，或者语言的语言的语言等等感兴趣的人来说，这份论文集无疑是一座蕴含了丰富资源的智慧宝藏。

□ Symposium on Extendible Language, *Sigplan Notices*, 4, No. 8, Association for Computing Machinery, New York, 1969.

所谓可扩展的或者可延伸的程序语言，就是其自身能够被程序员操作的语言。这次讨论会论文集相当成功地代表了其出版时的最高发展水平。

□ Sammett, J. E., The Use of English as a Programming Language, *Communications of the ACM*, 9, 3 (March 1966), pp. 228－30.

□ Higman, Bryan, *A Comparative Study of Programming Languages*, Section 15. 3: Subsets of English, New York, American Elsevier, 1967.

有一些人错误地认为解决一切问题的方法就是通过使用“真正的”语言进行程序开发，如果这些人能够读读这两篇评论，就会知难而退了。

① IFIP, International Federation for Information Processing，国际信息处理联合会。——译者注

第11章　程序语言

评　　注

多年以来，程序语言在某些方面变得更“自然”了，与此同时，我们好像也终于放弃了用“自然语言”编程序的荒唐想法。变化是双向的，正如许多程序语言中的词汇和概念如今也已在不知不觉中为自然语言所用。比如，尽管“信息倾泻”在软件开发中已经变得不常用了，但在自然语言中，“记忆倾泻”已经成为形容那些令人生厌的、独自滔滔不绝长篇大论者的一个术语。

在过去的25年间，我们都学习了不少的知识，但其中多数，我们都没有注意到。当我重读关于程序语言缺少“听说信道”的那段时，我意识到，在写作时，我从来没有跟盲程序员——那些整天使用“听说”程序语言的人——一起工作过。唉，对盲程序员而言，听说信道在不断改善；与此同时，程序语言正在向“可视”的方向发展——使得盲程序员的生活变得更加复杂了。虽然细节变了，但原则并没有改变：萝卜白菜，各有所爱。

面向对象（OO）的程序开发提供了一个相关的案例。尽管许多OO的爱好者没有意识到，但第一个OO语言其实早在25年之前就存在了。当然，从《程序开发心理学》的第一版中，你看不出这一点，因为当时我并没有把这方面的内容加进去。回到当时，我并不认为OO的思想对一般意义的程序开发有什么重要影响。1968年，我曾在奥斯陆与Simula的设计者们一起工作，而且我自己也是一个Simula迷，但当时我是从一个仿真软件开发者，而不是一个语言设计者的角度去看待它的功能特性的。

那时，我没能意识到Simula的潜能——部分原因是，我发现自己的学生中很少有人能够掌握这种新的思考方式。即使在今天，许多程序员依然未能掌握面向对象的体系，虽然随着比我更好的老师出现，这种情况可能会有所改观。

虽然科技进步了，但有些东西却一直保持不变。虽然黑板已为白板所取代，但他们本质的功能是一样的（擦除的速度仍然不够快）。虽然语法错误在个人计算机中所带来的后果轻多了，但程序员依然向他们的计算机祈祷。虽然程序语言的级别提高了一两个层次，但程序真正出错时，我们还得去分析16进制代码。

最后一个例子，“天才裁缝Levine”（或“天才语言设计者”）依然备受我的读者的青睐。我们依然不知道“语言设计者是不是天才，或者我们这些普通的程序员是不是瘸子”。

> “我们不知道语言设计者们是不是天才，或者我们这些普通的程序员是不是瘸子。”（p. 292）

第12章 程序语言设计的一些原则

前面已经说过程序语言设计的原则是靠经验得来的，下面我们将主要从理论的角度，对一些人所提出的有关优秀设计的一般性原则做一综述。之所以说主要从理论上来分析，只是因为大部分经验数据根本不存在。虽然那些数据并不一定很难获得——我们所能援引的大多数都是学生们在有限的时间和经费的条件下所作的研究。这里真正的问题是心理学因素尚未被纳入程序语言设计的研究中——或者说，即使被意识到了，心理学问题也被认为是可以通过语言设计者及其同伴的自省来解决的。

一致性

在有些情况下，我们可以从心理学的主流研究中获取实验数据，至少可以对可能存在设计难度的领域提些建议。一个例子就是**一致性**原则。众多的实验表明，记住一组数据的难度与其“信息量”有关。然而，“信息量”是很难测量的，因为它不仅依赖于给出的这组数据，而且依赖于没有给出的数据——记忆者脑海中存在的某些东西。举个例子，如果我们让一个参与实验者（被试）记住下面这组数

000 001 010 011 100 101 110 111

如果他了解二进制，这太容易不过了。

但假设被试了解二进制，而且被要求记下面这组数：

000 001 010 100 101 111

虽然这次只有 6 个数，难度却增加了。为什么呢？因为对于了解“系统”的人来说，记忆的复杂度与要记的数的多少没有关系，而是与系统外的数之多少有关。一旦知道了二进制体系，要记住前 1024 个数与记住前 8 个数一样容易。在第一个例子中，被试只需要记住“前 8 个二进制数”，但在第二个例子中，他可能这么记忆“前 8 个二进制数——除了 011 和 110”。即便不是心理学专家，我们也能看出第二个例子不比第一个容易，而且应该更难一些。

如果被试根本不知道什么是二进制，上面的结论就不对了，因为那样的话这两组数在他看来多少有点“随机”。这时，6 个数要比 8 个数稍微容易记一些——跟上面对高级被试的结论恰好相反。

在程序语言中，这种情形也随处可见。上一章我们提到 FORTRAN 下标的例子。如果程序员对代数表达式一无所知，我们可以说 FORTRAN 的下标规则比 PL/I 或 APL 好记，因为 FORTRAN 中可能的下标集合明显小得多。但程序员并非如此无知，而且他们仅凭一句话就能记住大多数 PL/I 或 APL 的下标规则，那就是：“任何标量表达式都可以做下标”。而在 FORTRAN 中，他们必须要学习下面 7 种规则：

```
a
v
v + a
v - a
a * v
a * v + b
a * v - b
```

其中，v 代表一个不带下标的无符号整型变量，而 a 和 b 代表大于 0 的无符号整数。

为了测试记住这些规则的难度，我们对 117 名资深 FORTRAN 程序员进行了一项调查。首先，让他们回答表达式

```
21 - K
```

是合法的 FORTRAN 下标，不是合法的 FORTRAN 下标，或者不能肯定。31 个人说是合法的，9 个说不合法，77 个说不能肯定。（作为对照，

我们问了 53 名资深 PL/I 程序员关于 PL/I 的同样问题，除了 4 个人都说是合法的；而这 4 个曾经是 FORTRAN 程序员）。

随后，我们请 9 位答对的 FORTRAN 程序员写出允许的 FORTRAN 下标。没有一个能准确无误地给出所有规则；而那 49 位答对的 PL/I 程序员则都能给出类似于“什么都行”的答案。尽管这并不是一个严格的规则——它包括了不合法的表达式，比如数组——但至少不会阻碍程序员使用合法的表达式，如 FORTRAN 中的 K − 21。如果 PL/I 程序员的确在下标中使用了数组表达式，编译器将会发现这个错误；而 FORTRAN 的编译器则不会警告不敢使用的 K − 21 的程序员，说他没有充分利用 FORTRAN 的能力。

虽然我们不能给出**一致性**原则的精确定义，因为它是一个心理学而不是数学原则，但从这个例子应该可以清楚地看出**一致性**的含义。因此在程序语言设计中一致性原则大致可以如此表述：“不管在哪里，同一件事情应该以同一种方式完成”。如果一门语言与该原则有出入，它就不符合一致性原则，因此将会更难学习，更难保证无错误使用，更难以用来创造“新”技术。

通过对使用两种语言——其中一个是另一个的子集——解决同一个问题进行比较，我们对这个问题进行了更深入的研究。这两种语言是 PL/I 的 OS/360 和 DOS/360 两个版本。DOS 版本并不完全是 OS 版本的子集，但很大程度上可以这么认为——好比是 OS 版本加上某些限制和省略（是限制的一种特殊形式）。每种语言有 14 个较新的 PL/I 程序员去解决同一个问题，我们发现，OS 组较容易写出能工作的程序而且代码长度短一些。在分析他们编写的不同程序时，我们发现有些其实可以用的特性 DOS 程序员往往没有使用。被问及时，他们总是说“我不敢肯定可以那么写啊!”（见图 12 - 1）。

索引问题			
	平均运行次数	平均运行时间	平均语句条数
完整的语言	32	1	243
该语言的子集	41	1.2	319

图 12 - 1　某一程序语言与它的一个子集的对比

一种语言的限制越多——即你不能做或说的东西越多，你就越会担心其他限制，从而缩手缩脚。即使限制出现在语言的其他部分，他们也会影响对当前其实无限制部分的使用。表述一致性原则的另一种方法是："如果一个程序员问，'我可以写……吗?'，答案应该是'可以'"。就好像经常被"不"打击的孩子那样，使用不一致的语言的程序员也会变得灰心丧气，不愿尝试新事物。

一致性的另一个重要方面是，同一个语法结构在不同的上下文不应该有不同的意思。比如，我们被告知，在 PL/I 中"不需要多余的括号，但如果你愿意，也可以使用"。这是个好规则，但不幸的是，它有几个很要命的特例。举个例子，下面这两个语句

```
CALL X (A);
CALL X ( (A));
```

的意思大相径庭，因为第二句总会导致传给 X 的参数为空（粗略地说，是传值而不是传引用）。另一方面，下面这两个语句

```
GET LIST (A (I) DO I = 1 TO N);
GET LIST ( (A (I) DO I = 1 TO N));
```

的意思也不一样，因为第一个在语法上是错的——在循环声明中数据表需要"多余的"括号。

这些语法上的不一致性无疑会打击程序员进行语义探索的积极性，因为它会导致程序员不清楚自己有多大的驾驭语言的权力。假如已经遭遇了括号规则的上面两种特例，程序员可能会不情愿地在一个复杂的表达式中加上括号以消除异义，或者，不敢使用多余的括号因为他必须"肯定"那样做没错。总而言之，他会感到困惑，而程序员本不应该被他们的程序语言搞糊涂。

作为对一致性问题的很有意思的补充，我们也许会注意到 PL/I 语言中的括号所引起的麻烦来自于有限的字符集所导致的括号的滥用。这种有限字符集的后果是很普遍的，因为无论何时，当同一个字符在不同的上下文中出现时，不一致性就随之而来。APL 在试图使用大字符集方面可能比任何现有的语言走得都远，甚至到了使用不同的符号来表示"减号"和"负号"的程度。APL 的支持者也因为这种一致性声称它是功能最强大的

语言。

当然，上面的结论应该是可以通过实验验证的。怎么验证呢？举个例子，我们来看图 3－1 中的那个冗长的，被许多括号括起来的语句——在 PL/I 中有 6 层括号。用 APL 语言来表达，这个语句形如图 12－2 所示。一眼看过去，感觉这个语句不像图 3－1 中那么复杂，一方面的原因是 APL 使用了两种类型的括号，另一方面的原因是，得益于 APL 的函数形式和自右向左的匹配规则，省去了两对括号。这种直观的感觉究竟对不对呢？我们的检验办法是生编硬造出一些这两个语句的错误变形，让资深的 PL/I 和 APL 程序员找出其中的错误。结果 APL 语句无论在速度还是准确性上都明显占优，从而说明了 APL 号称的一致性至少在某些方面还是很有根据的。

```
ANGLES[I]←2×ATAND SQRT (S-A[IND[I;1]])×(S-A[IND[I;2]])÷S×(S-A[IND[I;3]])
```

图 12－2　图 12－1 的一个 APL 版本

字符集的限制，仅仅是由于机器的局限性而导致不一致性的一个例子。FORTRAN 的下标限制是另一个例子，至少它导致了程序员神经紧张，认为如果不对下标加以限制，就会影响代码的效率。在追求其他的某些设计原则时，也可能会引起不一致——因为，这些原则的提出并不见得都有根据。没有人质疑这些最基本的原则，尤其是无二义性假设。连 Bryan Higman 这样老道的作家都说，“因为每种语言都必须有其字符集和无二义性的可判定的语法……”，但他丝毫没有怀疑无二义性假设。对如此根本性的假设，我们应该时常加以检查，以确保它不致（由于条件的改变而）造成什么危害。

说明无二义性会导致困惑的一个漂亮的例子，就是 PL/I 中的 WHILE 语句。在关键词 WHILE 之后，是一个逻辑条件表达式，正如 IF 语句中关键词 IF 之后的表达式一样。但在 WHILE 语句中，条件表达式必须用括号括起来，而在 IF 语句中则不必——这种不一致性将会导致许多语法错误。之所以需要这对表面上看起来有些多余的括号，是因为如果程序员声明了一个名为 WHILE 的数组并且使用如下语句：

```
DO WHILE (I) = 1;
```

这行代码在语法上就会有二义性了。就因为这个原因，每个使用 WHILE 语句的程序员都必须多用一对括号。

事实上，尽管语言设计者对二义性有着近乎病态的恐惧，它依然悄悄侵入了各种语言之中。但多数情形的二义性都是心理学意义上的——看起来好像不在设计者的考虑之中。乘除法混合就是一个经典的情形，如：

A = B/C * D

虽然语言设计者和编译器并不认为该语句有歧义性，程序员却认为有——我们发现的与此相关的一系列错误也证明了这一点。

针对二义性，当务之急是承认它可以分为两种——物理上的和心理上的。当然，编译器从来不会有歧义——一个编译器可能会把上面的语句解释为：

A = B/ (C * D)

另一个可能会解释为：

A = (B/C) * D

但每个编译器的解释都会始终如一。由于编译器的编写者可能会有不同的解释（最终真的会导致心理二义性），消除语言中的这种二义性是至关重要的，从而在实现过程中，只做必须要做的，不必要的就不要做。但是，仅仅消除物理上的二义性，还不能从根本上解决心理上的问题。

Ken Iverson 是 APL 语言的发明者，同时也是该语言的开发者。有一回他告诉我说，APL 语言的这种“自右向左”式的消除二义性的规则，要比诸如 PL/I 等语言使用的规则“更为自然”。从正式的意义上讲，PL/I 语言用来消除二义性的规则经常会使得计算执行的次序模棱两可。虽然相比之下，APL 语言的这种严格的自右向左式规则的表述形式显然要更简单，但是作为一种消除二义性的方法，其“自然性”依然是一个尚无定论、值得进一步研究的心理问题。按照自省的观点来看，我必须承认，即使是在编写过 400 段 APL 程序之后，我也没有完全习惯于这种自右向左式的思维方式。当然，我的那些只写过 10 来段 APL 程序的学生们，也肯定没有习惯。由于受到其早期形成的从左向右式习惯以及其他规则的影响，其心理上的二义性问题依然存在，在我所见过的 APL 程序错误中，这即使不是最主要的原因，也是最主要的之一。而就我本人而言，即使是

在使用PL/I语言时，我也已经习惯于通过哪怕是过量使用括号的办法来避免在心理上可能引起的二义性。

已经有迹象表明，程序语言的发明者会认识到这两类二义性之间的区别。其中Klerer和May采用的办法很有趣，他们允许程序员按照自己最习惯的方式来书写程序，然后程序员将看到这样的提示信息：

THIS IS THE WAY WE INTERPRET YOUR STATEMENTS.（我们按照这种方式来理解你的语句。）

IF ANY ARE INCORRECT PLEASE RETYPE THE STATEMENT CORRECTLY.（如果我们有错，请你将这条语句重新输入一遍。）

在终端环境中，这种方法尤其显得有趣——至少，我们有了某种接近自然语言对话的形式。这时我们需要重新回顾一下关于“语言”的固有观念。

对于那些（诸如PL/I语言中WHILE语句等）极为罕见的二义性，这种方法显得尤为有效。如果希望完全消除二义性的可能，那么我们为什么不干脆在语言中设置一条秘密规则，并且只在程序员使用了一个名为WHILE的数组时加以应用呢？在这种情况下，一旦出现这种WHILE-子句，编译器将会要求该程序员更换掉它；而同时，不会打扰任何其他人。当然，编译器在实际中遇到这类情况时，将会提示我们说它已经自动为我们插入了一些括号。为什么不给出警告信息，而偏偏要去惩罚那些懵懵懂懂地把关键字当作变量名的程序员呢？

这道出了我们关于一致性的最后一个问题。所有的责难都落到程序语言设计者的头上，有失公允——对这种过错，程序员个人也应该承担更多的责任。比如，要是使用了关键字给变量命名，该名字本身就意味着程序中具有不一致性；所以肯定地，其他人在试图阅读该程序时，必然会发生混淆。（我早期的一些书中所使用的例子，就属于这种情况，我对此深感内疚。）还有其他一些原因会引起不一致性：比如在声明变量时，在毫无特殊原因的情况下，设置的大小不一致；又如，在这里使用缩写形式，而在别处却使用完整的名称；再如，在书写源程序时，随意地缩格或空格；还有，在这里过于密集地添加注释，而在另外一些更困难的地方却没有1行注释。

然而，由程序员引起的非一致性的最主要原因，却是在选择助记的名称方面。亚伯拉罕·林肯就经常引用这个谜语："如果我们把尾巴称为腿，那么一条狗会有几条腿？"要是你回答说有"5 条"，那么林肯就会很吃惊地说："不，应该是 4 条——你**称之为**腿，并不能使不是腿的东西变成腿。"有多少次——有几千次——我们在程序中看到过：名为 OLDX 的变量并未被赋以变量 X 原来的数值；而名为 SUMAB 的变量也没有被赋以变量 A 和 B 的和；或者，名为 BIGGEST 的变量实际上记录的并非最大值？确实，**称之为** OLDX 并不能把它**变成** X 原来的值；但是无论我们如何频繁地发出警告，这类错误似乎还是持续不断地出现。正如程序语言自身中存在的不一致性一样，由于这类错误会令我们接受一些错误的观念。所以它们也会给我们造成损害。

从调试的观点来看，零星的不一致性要比大量的不一致性更糟糕，因为这时大部分地方的一致性会令人麻痹大意。如果在大部分时间内，OLDX 确实存储了 X 原来的数值，那么一旦出现不是的情况，我们将更容易遗漏掉；而如果在大多数地方，（A）的含义与 A 确实一样，那么我们可能将永远无法找到程序的错误到底在哪儿。

简洁性

我们已经提到过这样一个事实：在能力方面，人类的头脑有某些来自先天的局限。这些局限可能会因人而异，也可能对于不同的问题也不尽相同。尽管如此，我们还是可以肯定地说，对同一个人而言，一段短些的程序总是要比更长的程序要更容易理解。比如，通过对注释的研究我们发现，即使是添加了几行不会执行的文字——也即使这些文字是明确地为了**提高程序的可读性**——也会增加程序阅读的困难。

然而，为了衡量程序的简洁性，并不能简单地统计一下使用了多少字符来表述它。在我们需要计算输入错误时，字符的数目**的确**很重要。尽管如此，至关重要的单元通常都不止一个字符。例如有两个表达式：A + B

和 BIG + SMALL，理解前者的难度绝对不是后者的1/3[①]。之所以不是，是因为在思维过程中，我们极少会使用单字符来表示最基本的单位：我们更习惯于使用“单词”。众多的实验结果表明，诸如 BIG 等单词会比单词 A 更容易为人接受和处理。

人类具有一种能力——把若干个小单元合成一个大单元，将它们看作一个整体——这样，就很容易对它们进行处理。在心理学中，这种信息处理的能力被称为“组块”[②] 能力。组块是一个重编码的过程，通常我们都是在无意之间完成的。但是，尽管人们的组块过程是自然进行的，但是每种具体的组块能力却是需要通过经验才能掌握的。当第一次遇到诸如“110010111100”的二进制串时，我们不得不按照 12 个单独的数字来记忆。但是经过一段时间之后，我们就会掌握 8 进制或者 16 进制，这两种进制都或多或少可作为组块的正式方法。于是很快地，我就会习惯于立即把这个二进制串“看作”是“6274”或者“CBC”；这样，包含这类数字的问题就可以更“简洁”的描述出来。

程序中的一些组块方法，直接借鉴了人们在其他地方已经使用过的组块方法，然后自然演化而来，比如 DO、READ 和 ALLOCATE 等“单词”。而另外一些组块方法，则要等到我们初次开始在代码簿上书写程序之后，才能逐渐掌握的——因此，即使是编写同一个程序，由于程序员在程序语言方面经验的不同，其长度也会不尽相同。对于一位经验丰富的 PL/I 程序员来说，下面这条语句：

```
DO I = 1 TON;
```

可能只需一瞥就可以看懂。但是对于初学者而言，则不得不把分成六七个更小的组块。

使用组块方法可以让常用的字符串看起来更加简洁，但是程序语言的设计者却会受到一些限制，比如不能利用程序员在学习该语言之前已经掌握的组块方法，也不能利用程序员在使用该语言时偶然学会的组块方法。但是这些语言的设计者可以为同一对象的表达提供不同的方式，通过这种方法，程序员就有可能以不同的方法进行压缩。在 PL/I 语言中关键词的

① 两个表达式的长度分别为 3 和 9。——译者注

② Chunking，记忆或知识的组块，亦可称为“串节”。——译者注

缩写方式（比如可以将 PROCEDURE 缩写为 PROC），就是这类方法的最简单例子。关于自然语言，有一个基于经验的结论被称为 Zipf 定律，该定律指出：自然语言中最常用到的单词，其长度也会趋于最短。这种例子很多，比如英语中的单词“is”、“the”、“a”、“I”、“he”和“it”等。在英语中的缩写形式中也有此类例子，比如“don't”、“there's”等等，而最为通常的此类组块方法，是经过了长期演化过程才形成的，因此不易为人察觉。

但是在程序开发中，我们则可以更经常地看到这种正在演化之中的过程，比如，某位程序员可能从此一直用 PROC 来代替 PROCEDURE。但是对于那些很少触及的单词（比如 STRINGRANGE 等），尽管其长度要比缩写的某些单词要长很多，但是程序员可能根本不会去掌握它们的缩写形式。在一些函数被另外一些功能更为强大的函数取代之后，也会相应地提供另外一种缩略方法。比如在程序员学会使用函数 SUM（X）之后，他就可能再也不必使用 DO 循环语句组[1]，也可能再也不必先依次罗列出所有可能的情况，然后再使用 DO 循环语句组。当然，在编写程序时所有可行的压缩方法中，最基本的方法就是循环迭代——该方法最早由 Lovelace 夫人发现，而离开这种方法，现代的计算机就根本无法实现。诸如 SUM 等函数的概念也已经被最早期的程序员所了解，尽管就概念而言它不如迭代的概念重要，但是从心理和实践的角度来看，这些函数的意义或许更大。

当然，语言的设计者可以通过不同方式来提供函数功能：可以提供内置的函数，比如 SUM 或者 SQRT 等；也可以提供一些机制，使得程序员可以自己来构造函数。虽然现代的程序开发语言几乎都具备上述两方面的特性，但是它们实际上提供的机制在细节上迥然不同。使用由程序员定义的函数并非尽善尽美，其中最特殊的一项限制就是：虽然语言的设计者可以在其函数中实现种种功能，但是通过语言提供的机制，程序员却很少能够再现这些功能。比如，PL/I 语言中的内置函数具有很多强大的功能；虽然它同时也提供了强大的子程序定义的机制，但是在编写自己的子程序

1 Do-group，循环语句组。在 PL/I 语言中，一种以 DO 语句开头并以相应的 END 语句结尾的语句序列，用于控制程序的运行。——译者注

时，程序员却还是无法实现完全通用的同等功能。

另一类简洁性来自于数据操作的结构化——尽管应用的时间不长，但是它的威力最终或许会比迭代或子函数更加强大。比如在 APL 语言中，所有对单个标量的操作，都可以按照统一的形式应用于任意类型的数组。在 PL/I 语言中，提供了一些数组操作的可能，而同时也支持结构化的操作。这些操作功能之强大的一个体现就在于，使得繁杂的多个操作可以简洁地描述为一个操作。为了说明这一点，请考虑我们此前曾经提及的 OLDX 问题。如果不是按照 OLDX、OLDY 等等的方式进行变量声明，而是统一地声明为一个结构：

```
1    OLD,    2 (X, Y, Z, ...)
```

那么我们只要执行一条语句

```
OLD = NEW;
```

就可以把所有旧的数值替换为对应的新数值。

有许多专用的程序开发语言提供了更高级的数据结构，由于通用语言不支持这些数据结构，所有前者的功能更为强大。只要这些特殊的数据结构与用户的需求一致，就可以实现多样化的压缩；反之如果这些数据结构与用户的需求不符——比如使用 LISP 或者 SNOBOL 的用户哪怕是需要进行简单的算术运算——这些结构反而会起相反的作用。为了从这种困境中解脱出来，最有希望的方法就是在诸如 PL/I 等通用语言中嵌入特殊的数据结构。为了使得程序员可以在语言本身之外，自行地构造数据结构的处理系统，PL/I 语言确实提供了一些机制；但是同样存在上面提到的困难，即无法再现语言本身提供的通用功能。

使用适当的数据结构实现压缩的另一种办法，是通过低级的操作来获得高级的结果。例如，在顺序处理一个已经完成排序的文件时，由于可以假设所有的记录已经按照次序排列，所以我们可以快捷地判断出是否有记录缺失；否则（文件没有事先排序），为做出这种判断，就需要很多繁重的程序开发工作。另一项进一步的改进是，允许把某个结构的数值整体赋予另一个结构——例如为了设置一个串中的所有比特位，我们只需一次性地对整个串进行赋值，而不需要对各比特位逐一赋值。这种方式的操作把对数组和结构表达式各成员的重复性操作隐藏了起来，这种整体的操作描

述与其隐藏的操作之间的区别，正像语言本身的抽象定义，与其在不同计算机上的具体实现方式之间的区别一样。在适当的硬件环境中，可以直接对数组进行操作——这就和我们现在对串结构的整体赋值一样。

在结束关于程序压缩的讨论之前，我们还要花些时间对问题的另一面作一考察。或许，终极的压缩形式就是完全消除各种特定的情况——通常，也就是最经常使用的那种情况。比如，在 PL/I 语言中，我们会把 SKIP（1）简写成 SKIP，因为截至目前这是最为常用的情况。而在定义一个变量的时候，我们可以通过默认赋值的方式，为它设置一些属性值。比如下面简单的声明：

```
DECLARE      X;
```

其含义与下面的复杂形式是等同的：

```
DECLARE X FLOAT DECIMAL(6) REAL AUTOMATIC INTERNAL;
```

而实际上，在很多情况下，整个声明都可以省略掉，这时的变量 X 就是隐式声明的或者上下文说明的。这两种极端情况之间的差别，就像中文与英文语法之间的差别一样。英语中有很多冗余。当我们说：

He went two time（s）.

其中表示复数的后缀 s 就是多余的，因为前面的“two”已经明确地表明了这个意思；而在中文中，就没有任何与这种附加字母相对应的语法成分。在正常的口语交流之中，这种冗余确实非常有用——因为即使语言的一部分在传播中可能会走样，听众仍然可以通过这些冗余，准确地领会说话者的意思。但是在书面写作中，这些冗余就没有多少价值，而更多地成为一种负担。因此，很有必要在所有程序语言中，引入一种有意义的默认值赋予机制。

但是，能否使用这类默认值系统，取决于是否存在一个受限的语言范围——至少，这种语言范围的受限程度必须是可以满意的。也就是说，如果确定的默认情况出现的可能很小，那么这种做法就不会对我们有多大帮助。随着可能出现的情况越来越多，为了确定到底哪种情况才是默认的，我们需要付出的代价将会越来越大，到最后还不如重新定义一套完整的语言规范更为省力。而要是各种情况出现的概率不相上下，那么使用默认值系统的机会就会很小，于是这种方法就将变得没有意义，因为此时默认系

统不值得去实现。相反地，如果大多数情况都采用了默认值，那么在必须处理那些非默认的情况时，我们又将费很多时间去回忆。虽然还没有通过实验对这一问题进行研究，但是我们还是有理由猜测：在相当程度上会经常被用到时，默认值方式才会变得最有用；而如果是过于频繁地被用到，其作用反而会下降。

当然，只有对使用该语言的特定程序员，上述原则才可以适用。而在多种不同类型的程序员使用这一语言时，默认值的办法非常有用。例如，之所以使用 REAL 的默认类型，仅仅是为了与 COMPLEX 类型相区别；而或许有九成以上的程序员根本都不知道后一种数据类型，更不用说使用了。通过与此类似的默认值方法，同一种语言可以演变成多种不同的语言——因为从效果上来看，那 90%不懂 COMPLEX 数据类型的人所使用的，正是一种无法处理复数运算的语言。

然而，只有在对许多细节问题加以考虑之后，这种通过默认来自动替换的方法才会行得通。比方说，在 PL/I 语言中，提供的内置函数对不同类型都是通用的，也就是说，这些函数会根据其被调用时使用的参数类型，自动地选择适当的操作符[①]。如果我们执行 SQRT（X），那么对于实数类型的 X（X REAL）和复数类型的 X（X COMPLEX），其具体操作是不同的。对于 REAL 类型的 X，X = −1 将会使导致程序错误；而若是 COMPLEX 类型的 X，就会正常计算，其结果为 1I。那么，当 X 是 REAL 类型时，为什么不同样给 X 赋值为 1I 呢？请考虑一下，要是有某位冒失的程序员写下这样一条语句：

```
A = SQRT (X);
```

但是他对 COMPLEX 类型一无所知。如果右侧的计算结果为 1I，那么由于 1I = 0 + 1I，所以只有实部（0）将参与赋值，于是变量 A 将被赋值为 0。因此，如果程序员缺乏关于复数的知识，那么这类错误将很难发现；而如果默认的是在实数的语言子集中工作，就可以把这类程序员保护起来。

这样，通过适当地消除冗余，我们也可以同时避免程序开发过程中的

① 在后来的诸如 C++等面向对象的程序语言中，这种方法被称为“重载”。——译者注

乏味，甚至减少错误的出现。最极端的情况是，程序语言中所有可能的冗余都已消除——在这种时候，任何可能的字符串都对应于一个在语法上正确的程序，因而根本不需要对程序做任何语法检查。某些系统中采用的机器代码，就是这样的例子——在这种语言中，任何可能的操作码都是允许的，而任何地址的指定都是合法的。我们知道，使用机器语言进行程序开发是极其困难的工作——那么，造成这种困难的原因，是不是由于在消除冗余时矫枉过正呢？

为了回答这个问题，我们必须明确地把在心理上有用的冗余，与在心理上无用的冗余加以区分。有这样一个例子能很好地说明这个问题——请考虑这样一条 FORTRAN 表达式：A（1+I）。要是 FORTRAN 语言禁止这个字符串，那么编译器就会发出错误提示。既然并非所有的字符串都被允许，那么就可以通过对非法字符串的检测，来实现错误检查。但是编译器在这里究竟检查了什么呢？之所以要制定禁止 A（1+I）的规则，不是出于对程序员心理舒适度的考虑，而是为了能够更方便地编译出“优化的”代码。尽管这个表达式确实是错误的，但是该错误与我们的问题毫不相干，而只是由于 FORTRAN 语言的限制。所以从某种意义上讲，出错的并不是程序员，而是语言本身。

局部性与线性

在先前的讨论中，我们曾经给出过一个例子，用来说明即使没有程序清单在手边，程序员仍然可以借助出色的记忆力来记住相关的信息。实际上，与良好的记忆力一样，精心设计的程序语言也可以尽可能地让有关的信息唾手可得，从而同样使得程序员如鱼得水。

在这方面，我们对人类的两种记忆能力感兴趣：所谓的“联觉”（synesthetic）和“顺序”（sequential）记忆。正是凭借联觉记忆能力，我们才可以识别人脸、邻居或是书中某一页的版面布局——而不必去追究特定的细节。该单词来源于希腊语——“syn”意为“共同地”，而“esthetic”意为“感觉”或“感触”。因此所谓的联觉记忆能力，是我们记忆力中的一部分，它可以使我们在匆匆一瞥之间，把事物作为一个整体来

记忆。

而顺序记忆能力则更多地与听觉相关——比如在依次听过调音笛的前面 9 个音之后，我们就可以把第 10 个哼唱出来；但是如果直接被要求唱出某个音，就会困难得多。我们的顺序记忆的构成方式，似乎类似于方向不定的链条——如果你不介意的话，这就像语音信号一样——沿着这根链条，每个环节的出现会使我们回忆起下一个环节。在背诵诗歌时，或者有时当我们记忆一张列表时，我们使用的就是顺序记忆能力——随着我们朝一个目标不断前行，我们可以回忆起沿途的情况。

就程序而言，“局部性”的概念对应于联觉记忆，而“线性”概念则对应于顺序记忆。所谓的局部性，就是指我们可以在同一个地方——比如说某一页中——找到程序中所有相关的部分。要是没有良好的局部性，察看程序清单的程序员就不得不反反复复地在不同页面之间来回奔波——除非他的联觉记忆力超乎寻常，以至于可以记住分散在不同页面上的相关信息。程序语言鼓励局部性的方法之一，就是通过压缩——也就是说，如果整个程序可以在一页中容下，那么显然任何相关的部分都将唾手可得。

局部性也会在其他方面受到鼓励与排斥。能够使用文字将很有帮助，因为我们可以通过其名称来获得所有希望了解的东西。精心选取并使用助记符也会达到一样的效果，这使程序的读者不必翻到变量声明的地方，就可以了解其实际含义。自然地，如果一个变量只是在局部被使用，那么如果程序语言允许我们就近定义它，也会有助于阅读。然而在有些语言中，通常限制编译器只能对程序进行一趟扫描，这就要求在程序的开始处对所有变量统一进行声明。在这种情况下，为了阅读一段长程序，我们就不得不来回察看前面的页面，忙得不亦乐乎。

然而，尽管有的语言强制要求所有变量的声明必须集中到同一个地方，我们还是可以利用某种局部性——比如，我们可以把第一页撕下来，并始终把它与我们正在查看的页面并列地摆放在一起。实际上，尽管诸如 PL/I 等语言非常自由，允许在任何位置做变量声明，但是却反而因此招致了许多人的抱怨。与其他的自由性一样，程序员们之所以滥用这种自由性，是由于他们未能真正了解提供这种自由性的目的。在理想的模式中，每个 PL/I 程序都需要把所有的全局变量声明集中到一页之中——或者在

第一页，或者在最后一页——而所有的局部变量则必须在使用它们的代码所在小节进行声明。

为了说明非局部性可能引起的问题，请考虑如下典型的循环语句情况：

```
DO I = 1 TO 10;
PUT LIST (I, F (I)); END;
```

其中，F 是一个由程序员定义的子函数，其内容可能是：

```
F: PROCEDURE (J) RETURNS (FLOAT);
   S = 0;
   DO I = 1 TO J;
     S = S + I;
     END ;
   RETURN (S);
   END F;
```

由于变量 I 的作用域并非只限制于该子程序内部，我们将发现，主循环只会给出 5 组输出数值：

1　1　3　6　5　15　7　28　9　45

而不是像程序员很可能期望的那样，给出 10 组输出数值。而程序员之所以期望得到 10 组输出数值，是因为他误以为变量 I 被限定在 DO 循环的局部——没有任何在视野之外的东西会影响到变量 I 的取值。但是正如所发生的事情，变量 I 在函数 F 中被使用，但并未在 F 中进行声明，因此每进入 F 一次，I 的数值就会改变一次。（请注意，当 DO 循环

```
DO I = 1 TO J;
```

正常结束时，变量 I 的数值等于 J+1。）

通常，这类程序错误会造成极大的麻烦——因为当我们无法找到错误时，常常就是由于我们查找的位置有误。在这个例子中，我们就是找错了地方——我们很自然地会作出假设，认为只有那些在我们鼻子底下可以看到的地方，才有可能影响到变量 I 的数值。因此无论使用的是何种程序语言，在调试的时候，我们必须首先了解清楚有哪些非局部的因素可能会起作用。比如在 PL/I 语言中，（使用全局变量或者“通过地址”

指定的参数进行）子程序的调用、接通单元（类似于无参数的子函数调用，但是根本不需在局部作定义，就可以直接调用）以及通过一个变量来定义另一个变量，都可能导致一些微妙的效果；而仅仅考察程序中问题症状出现的局部，并不能找到问题的根源。

在程序语言中之所以会存在导致非局部性的功能，通常并非是由于其设计的不善；否则，人们早就会把它丢到垃圾堆中，而没有任何可惜之情。比如通过某个变量来定义另一个变量的功能，就是为了实现压缩特意提供的，而压缩本身也可以提高程序的局部性。因此，为了防止由于程序开发不善或者错误而引起的非局部性，在语言设计方面下的工夫应该适可而止。比如，在上面提到的例子中，PL/I 语言的设计者们也许本可以采用更好的办法，比如除非程序员专门说明，否则子函数 F 中的变量 I 或 S 都将被自动地默认设置为局部类型。当然，如果程序员确实对变量 I 或 S 进行了明确的声明，比如：

```
DECLARE I FIXED BINARY, S FLOAT;
```

那么只要在子函数 F 内部，该声明都将被限制在 F 的作用域内。通过这种方式，程序员保持对所有变量进行声明的习惯，哪怕他只是简单地声明：

```
DECLARE I, S;
```

就可以克服这种困难。

PL/I 语言提供的接通单元功能，是非局部性的另一个有意思的情况，因为接通单元的基本功能就是为了让我们实现线性。通常使用接通单元的目的，是为了对意外情况进行处理，而在有些程序中根本不会出现这类情况。但是，实际上从这个意义上讲，也可以被看作一种实现附加局部性的途径。首先，由于 PL/I 语言允许我们不必明确地指出条件的名称，所以可以进而使我们实现对代码主体的更有效压缩。其次，由于我们可以避开各种例外情况，将精力集中到更通常的情况上来，所以也会产生统计上的压缩效果。这样，接通单元的功能使我们可以忽略各种例外情况，从而简化了程序开发——然而这也不是没有代价的，因为这时我们需要时常回忆接通单元的设置。

无论如何，接通单元还是可以帮助我们实现程序的线性，这种线性将进而有助于我们的顺序记忆。关于那些类似于程序开发环境中的问题求

解，已经有一些实验，这些实验的结果表明：严格按照线性顺序编排的一系列判断，要比包含分支或循环的判断序列更加容易处理。有关程序语言方面的经验也似乎可以证实这一点，因为包含大量 GO TO 语句或者其他分支的程序总是由于难以理解或调试而臭名昭著。这种困难的部分原因要归咎于其非局部性，因为分支转向的地址很可能位于不同的页面上——比如对总共 50 页的程序而言，这类分支就将导致大量的翻页动作。

而即使分支转向的地址就在同一页面上，也仍然会引起混淆。首先，很有可能有其他的分支也会指向这同一个位置，但是这种分支却没有被程序员察觉，所以我们不能很容易就搞清楚，在执行过程中，哪些时候会到达现在的位置。其次，每次遇到分支，我们对程序的正常的顺序扫描或思考过程都会打断。纵使我们有时可以不必翻页，但是由于我们自然的进程被中断，所以更容易出错。

许多高级语言具有一些很好的特性，可以帮助我们屏蔽掉通常的“程序存储式”计算机中的非线性。其中的 IF 语句允许其后继执行的语句紧接在 IF 之后；如果在 THEN 或 ELSE 子句之后有多个需要执行的语句，那么就可以用 DO 或者 BEGIN 把它们组合起来；如果 IF 语句中条件包含多个判断，就可以用逻辑连接符把它们连接起来；通过循环式的 DO 语句，我们可以消除掉显式的分支；而对数组的整体操作，也可以为我们消除掉显式的 DO 循环；使用接通单元，使得我们可以处理意外情况，而甚至不必去提及或者想到这些情况——所有的这些以及其他的特性，都可以帮助我们消除程序中的非线性。在 PL/I 以及其他的语言中，都会通过给语句编号，以支持跳转功能——实际上，许多程序员都把这看作为程序结构不良的一个特征。虽然我们不希望就此问题继续深入讨论下去，但是有一点确实很有道理：GO TO 语句是应该从 PL/I 语言中清除出去的一条“主要”语句，程序员不会因此而有多大的损失。

传统与革新

为了使得程序开发语言具有轻松的表达方式，同时把错误率降到最低，一种最重要的方法就是是程序语言变得“自然”。正如我们已经看到

的，实现语言自然性的方法之一，就是借助一致性；然而，只有在程序员对该语言具有一定的经验之后，才有可能应用一致性。在刚刚开始使用一种语言的初期，我们还不能察觉到一致性——我们只能注意到，在语言中有一些成分似乎与我们预先对“正确”的感觉不符。我们的这种感觉来自何处？也许，这种感觉来自两个方面：一是自然语言，比如英语、法语、中文；二是程序开发语言，比如 FORTRAN、ALGOL 或者 COBOL。——在学习新的语言之前，我们对这些语言可能已经掌握了。如果能够使得程序语言在某种意义上与用户已经熟知的其他语言相一致，那么我们就可以使该用户更为容易地进入到陌生的新（语言的）世界中来。

我们可能把自然语言中一些概念搬到程序语言中来。作为这方面的一个实例，让我们来考虑一下有效空格的例子。在欧洲语言系列中，至少在正规的拼写形式中，有效空格的概念是非常自然的。但是在 FORTRAN 中，就没有这类与有效空格对应的成分——在文字串中的情况除外。

此外，正如分界符或者空格的概念，“多个空格与单个空格等价”的观念从任何角度看都是自然的。单词“orate”（大言不渐的演说）的意思与两个单词“or ate”（或者吃掉了）的意思绝对不同。但是如果省去其中的空格，通常还是可以矫正的——这是得益于有冗余的成分的存在，所以马上就可以检查出一些语法的或者语义的错误。但是，这并不见得一定保险，比如：

“He gave it to you, or ate it.”（要是他没有把它给你，就自己吃掉它了。）

“He gave it to you, orate it.”（他把它给了你，并且装腔作势地讲话。）

如果是由于其中的空格被遗漏而使得句子变得没有意义，那么在这种情况下读者还是能够通过有意识地把空格插回去，自行把它更正过来，比如：

“He lost it or ate it.”（要是他没有把它弄丢，就已经吃掉它了。）

“He lost it orate it.”（他把它弄丢了，并且装腔作势地讲话。）

但是，如果遗漏的空格是与一个分界符相邻，那么这种遗漏通常不会引起

人们的注意，比如：

“Leave it alone，will you!”（请别动它，好吗?）

倘若一种程序语言与自然语言中这些有关空格的规则不一致，那么就更难于掌握，或者至少会更难于调试。比如在 FORTRAN 语言中，下面这条语句就代表了一类典型的错误：

DO 33 I = 1.20

在这里，也许是由于穿孔错误，其中的逗号被错误地写成了句号，于是这条语句就会被理解为：

DO33I = 1.20

这里的空格给我们造成一种错觉，认为其中的 DO、33 还有 I 分别是 3 个独立的语句成分；但是实际上，根据 FORTRAN 有关空格的语法规则，他们仅仅构成了一个变量：DO33I。

在 OS/360 操作系统的任务控制语言中，也会由于其语法规则导致空格的处理不当，而且这类错误的代价要更大。这种语言与汇编语言的一个类似之处在于，操作数域的范围是由第一个空格界定的。之所以采用这种奇怪的习惯，原本是为了能够像 System/360 汇编语言一样，可以在操作数域之后直接写注释。简单地沿用汇编语言的这个思路，虽然会使得 OS/360的设计者们感觉习惯一致，但是实际证明，对于其他的用户而言，这是一种蹩脚的做法。首先，FORTRAN、COBOL 和 PL/I 语言的用户对这种习惯并不熟悉。其次，没有人会按照在汇编语言中的方式，在任务控制程序中添加注释。最后，在任务控制语言中，要比汇编语言更加频繁地使用到连续卡片；而在汇编语言中，只有在处理宏的时候才会经常使用连续操作。

由于这种考虑不周的做法会导致错误，下面就是一种典型的错误：

//XX　DD　DSNAME＝ABC，　DISP＝（NEW，KEEP），　C

在穿孔时，程序员无意间在‘ABC’后面的逗号后多打了一个空格。因为无论是在英语还是在 FORTRAN 中，这都不妨大碍，所以程序员根本没有察觉这个问题。但是，计算机却会察觉到这个空格，并且会认为在其之后的内容都是注释。于是，DISP 参数就完全被忽略掉了；而因为默认的参数是 DISP＝（NEW，DELETE），所以程序员将发现其数据集无法保

存。还可以构造出更为糟糕的情况，而这种由设计失误导致的错误，到底给计算机机时和程序员的时间造成了多大的损失，难以用金钱来衡量。当然，从效果上看这并非是设计的失误，因为在设计观念上，任务控制语言与 PL/I、FORTRAN 或者 COBOL 并不一样。由于任务控制语言是操作系统的一部分，所以人们根本没有把它当作是一种可用来“程序开发的”语言，这就是为什么有人在设计它时，直接就近借用了一些还算令人满意的技术——却带来了灾难性的后果。(请同时参看图 10 - 1 和图 10 - 2)

程序语言与自然语言可能更加一致的另一个地方，就是拼写。在阅读自然语言的文字材料时，如果遇到拼写错误，我们总是可以跳过去；即使拼错的单词毫无意义，我们也会不去留意。而在口语中这种灵活性就更大了，无论是亚特兰大市民说“pie”，奥马哈人[①]说“milk”，还是波士顿市民说“yard”，我们都能听懂。然而程序语言则不然，在拼写方面，它们显得过于迂腐；尽管很多时候我们想要表达的意思再明显不过，它们仍然要咬文嚼字，其功夫堪称登峰造极。缩写，当然也是另一种拼写的方式；但是与正确的拼写相比，缩写的概念可以延伸得更远，甚至像某些商业系统所做的那样，跟猜测挂上了钩。

毫无疑问，自然语言中一般的“不严格性”比比皆是，拼写上的随意性只是其中的一例。我们既可以说：

“Bring it here, will you?”(请把它捎到这儿来，好吗?)

也可以说：

“Will you bring it here?”(你可以把它捎到这里来吗?)

没有人会认为二者有甚区别。虽然程序语言中不是没有一点回旋的余地，但是的确远不如自然语言那样灵活。COBOL 语言中的“噪声字”就是一个灵活性的例子，这使得程序员可以更多地体会到与自然语言一致的感觉。而在 PL/I 语言中，在一些地方允许对语句各部分的次序进行重组，比如：

```
DO I = 1 TO N BY K;          DO I = 1 BY K TO N;
```

或者

① 奥马哈人，内布拉斯加东北部的印第安人。——译者注

DECLARE A FIXED BINARY;

DECLARE A BINARY FIXED;

以及

PUT SKIP LIST (A);　　PUT LIST (A) SKIP;

如果一种语言允许两种可能的表达方式，而且每次只能选择其一，那么该语言就更容易掌握——因为不同用户可以自由地选用**对自己来说**最容易或者最自然的方式。但是在这个过程中，也许他根本没有察觉到自己做了什么选择。一位老师如果习惯于其中的一种方式，那么他的学生可能根本不知道还有另一种形式，因为老师可能没有教过。或者，学生也许会认为这两种方式在语义上有所区别——于是，既然已经掌握了其中一种，那么他就会对使用另一种形式有所顾忌。

二者之间确实有可能会存在语义上的差别。但是，这种差别非常之小，以至于程序员在正常的生命期之内，很难对此有所察觉。下面的两条PL/I语句：

DO I = 1 TO N BY K;　　DO I = 1 BY K TO N;

并不是精确的相等。即使是这样的两个表达式：

(A=2) & (B=3)　　(B=3) & (A=2)

也不尽相同。二者在人们的心理上不尽相同：不同人之间有所不同；或者在书写和阅读的过程中，人们的感觉也有差异。姑且不论来自心理上的这些差异，二者在性能方面也有可能不同；这种性能的差异往往隐藏在程序背后，因此如果程序员了解到这一点，也许就会在利用上述灵活性时变得更慎重。

在一种语言的内部，有些差异是巨大的，而另一些则可能微不足道。类似地，在不同语言之间，也会有一些差别是不值一提的——或者，至少是不甚重要，以至于不值得去留意。比如在不同语言中，可能需要使用下面两种不同的表达方式：

DO I = 1 TO N BY 3;

或者

FOR I:= 1 STEP 3 UNTIL N;

但是，任何称职的程序员都可以几乎不费任何力气，就可以从一种方式过

渡到另一种。这两种表达方式之间的差别的确微不足道，因而很容易在同一种程序语言中共存。在有关不同程序语言之间相对优点的争论中，我们时常会看到，人们在一些鸡毛蒜皮方面争得不可开交——因为只有在这些方面，才可以对两种语言做直接的比较。或许会有那么一天，有人会心血来潮地做个实验，以确定在这种情况下，DO 和 FOR 到底谁才是“最好的”关键字——但是仅仅凭借这点事实，恐怕对上述争论还是难有多大影响。

在设计新的程序语言时，如果能够使之与另一种已经广为使用的语言近似，那么由于我们可以尽可能地保持此前的习惯，所以前期的学习速度就会大大提高。但是，我们还是要为这种前期的快速学习付出代价，而且是以一种有趣的方式，因为毕竟没有哪两种程序语言是完全一样的——否则他们就不能称为“两种”语言。如果两种语言的差别很是**细微**，那么就会引发一种被称为“抑制”的心理现象。这种抑制作用有两种形式：反馈的和前摄的。因为在内容上与此前已经掌握的知识类似，可能会对后来的学习过程产生影响——这种情况，就是所谓前摄的抑制作用。比如，熟悉 FORTRAN 语言的程序员，在学习 PL/I 语言的诸如 F 或者 E 等格式项时，会遇到巨大的困难。其原因在于，尽管二者非常类似，但是在 FORTRAN 使用句号的地方，PL/I 却使用逗号。如果翻看一下精通 FORTRAN 的程序员的 PL/I 程序，我们就常常会发现诸如 F（10.2）的格式项——而对其运行结果的错误，精通 FORTRAN 的程序员总是会感到一头雾水。

实际上，正是两种语言之间的这种**相似性**导致了这种抑制效应。通过对 FORTRAN 程序员使用的帧型的分析，也可以得到同样的结论。在学习编写 PSSSSSSSV. 99时，这些 PL/I 语言的初学者遇到的困难很少，甚至没有。这是因为在 FORTRAN 语言中，没有可以与之比较的结构。（不过，使用 COBOL 语言的程序员自然就会遇到麻烦。）反馈的抑制效应同样取决于语言之间的相似性，不过它是对以前已经掌握的语言发生作用。因此通常而言，原来精通 FORTRAN 的程序员后来一旦又学会了 PL/I，那么他就会发现自己很难再回过头来写 FORTRAN 程序了——而这并不仅仅是因为时间的流逝。

因此，如果要使两种语言变得相近，同时在有些特性上又互相有别，那么抑制效应就是我们需要为之付出的代价。既然不可能完全一样，那么就不如干脆更为明确地让二者有所区别，从而降低抑制效应的影响。但是，为了使后来的语言与早前的语言类似，除了抑制效应之外，我们还需要付出更多的代价。老式的结构也许就是不能适应我们正在从事的工作，这种情况下，语言在整体上的一致性就会被破坏。这时，受到早期学习经验的影响，人们在后来（对更新的、更强大的技术）的学习中，就会进展更加迟缓。换而言之，我们无法要求新的技术与早先的语言相符，这种强行实现一致性的“削足适履”式的做法，到头来只能是徒劳无功。

不同程序员之间的差异非常大，因此我们应该清楚：在使语言与程序员已经掌握的概念与结构相一致时，我们能到达的程度必然是有限的。自然地，虽然似乎我们可以指望所有的程序员都会认可一些简单的形式（比如数字和简单的算术表达式），但是我们很快就会发现，不同人都有各自不同的行事方法。松散性是程序语言支持个体之间差异的一种方法——它允许人们在内置的两个或者多个方案中自由地挑选。因此在某种意义上，一种松散的语言更容易适应不同程序员各自的偏好。

不过，为了使语言具有这种适应性，还有其他的办法。这些方法的共同特点就是有所创新，这些新东西中有的是针对个别的程序，有些则可以在一大类程序中得到应用。比如，我们所熟知的程序语言的定义函数或者其他的子程序的能力，就是这样的一个例子。程序语言这方面能力的另一个例子，就是支持对新数据类型的定义——比如说，结构和列表。有些语言支持对操作符的定义与重定义，但是归根到底，这只不过是函数的另一种形式。最后，也有一些语言的元语言是精心设计的。这些元语言可以提供诸多便利，比如编译时的方便，或者在程序执行过程中动态生成代码。元语言的采用，将会从根本上改变语言整体面貌。

通过借助这种便利之处以实现适应性，我们还能够在一定程度上实现与自身特质的一致——而这是无法直接依靠语言的设计者来实现的。只不过，随着我们越来越接近自己特有的思考和表达模式，我们同时也将离其他人的模式越来越远。程序开发并非只是一个人与一台计算机之间的交流

过程。其他人也可能需要阅读或者理解一段程序；然而要是采用适应性机制的成本过高，那么在阅读别人的程序时，你就会感觉犹如是在理解精神分裂症患者的语言。

在特定的计算中心里，具有一定适应能力的语言，将更容易通过调整，精确地满足当时具体工作的需要。当然，如果这种重定义工作超出一定的范围，那么实际上就等同于在定义一种全新的语言——而这种工作绝对不是业余程序员可以轻松完成的。如果正在使用的语言确实与计算中心的实际需要相去甚远，那么在动手进行修改**之前**，先寻找一种现成的、更接近实际需求的语言或许是更好的办法。可以肯定的是，我们有足够多的语言可供挑选。

专用的、多用的以及玩具式语言

在某种程度上，我们关于语言的讨论更加偏重于专业程序员所使用的语言，对业余程序员的需求只是略加评论。但是这里设计的原则却是深层次的，这些原则同样适用于业余程序员所使用的语言，只是强调的**重点不同**。如果要为某个特定的应用领域设计一种专门的语言，我们只要对讨论的问题加以限制，就可以立即在心理上获得回报。于是，一种用于简单统计的语言，就不再需要复杂的算术运算功能；而对于那些用于字符串处理的语言来说，算术运算功能就完全是多余的；而任何一种机床控制语言，都会对几何、而不是算术计算能力有更高的要求。然而有时候，之所以有种种这类限制，仅仅是因为首先对语言的能力做了限制——这时人们已经假设：这个领域的问题绝对不会超出某个范围。于是，用于模拟的语言就可能不需要使用附加存储空间的能力，这意味着完全能够摒弃语言原来所提供的众多功能——当然，这同时也意味着，一些特定的问题可能会因此无法进行模拟。

在设计专用的语言时，对语言涉及问题的范围进行限制，是极其重要的。因此这项工作首先需要，而且必须经过深思熟虑。比如说，APL 语言之所以会取得如此巨大的成功，不能不说是得益于对其所需要处理问题的范围的精心限制——尽管大多数使用 APL 语言的程序员

仍然认为 APL 是一种“多用途的”语言。但是即使是对语言做简单的限制，使之适应终端环境、相对更小的工作空间，或者无需附加文件，语言设计师也可以为问题求解提供**出类拔萃**的工具——当然，是在其领域之内。这种语言之所以成功的另外一点原因，是得益于相当数量人的问题确实属于这个领域——或者至少在他们学习 APL 语言时是这样。但是如果程序员需要进一步解决诸如数据处理等问题，需要处理大量的数据，那么他们很快就会感到使用 APL 有点“力不从心”。许多人热衷于该系统，但是他们大都没有意识到其中的种种限制，并且正是由于这些限制才使之获得成功。

在非用户的眼中，这些限制是显而易见的，但是在一般用户的眼里情况则不然——其中的一个原因就在于，程序语言的确会反过来影响人们的思考过程。这里讨论的不是简单的抑制效果，而是对处理程序和数据的组织结构在大范围上的影响。我曾经教过一个 PL/I 语言的初级培训班，其成员为原来使用 COBOL、FORTRAN 或基本汇编语言的程序员，还有一些根本没有任何程序语言基础的学员——在教学过程中，我最明确地意识到了上述广泛的影响。在阅读学员们提交的第一次作业时，我惊异地发现，自己完全可以猜测出不同程序员原来的背景。我把这些猜测记录下来。当时我的手头上并没有这些程序员的背景资料，也就是说我们不可能有其他的方式来了解他们的经历。在第二天的课堂上，我对自己的猜测进行了核对，其结果如图 12－3 所示。

猜测 / 事实	COBOL	FORTRAN	BAL	无基础	猜测正确率
COBOL	6	0	0	2	6/8
FORTRAN	0	5	0	0	5/5
BAL	0	1	6	1	6/8
无基础	1	0	0	5	5/6

图 12－3　对学员背景的猜测

可以肯定的是，这些猜测都是简单地根据特定的线索做出的，比如：

在 F一格式项中使用了小数点，而不是逗号。但是，这些特定的线索有几回也令我做出误判。比如我把一名使用基本汇编语言的程序员误认为是使用 FORTRAN 的，因为他也在上述地方使用了小数点；我也曾把一名没有任何经验的程序员误认为是使用 COBOL 的，因为他使用了 PART _ COST 等符号，其中的下划线是 COBOL 中的连字符。（实际上，他是在见到另一名使用过 COBOL 的程序员这样做之后，才开始模仿这种风格的。）

程序语言会对人们的思考过程产生影响。其中一个后果就是，在人们试图去解决某个问题时，会受到了解哪种程序语言以及了解到何种程度的限制。于是，就会出现一种对专用程序语言以及自己目前知识水平的自我满足的倾向。专用语言会以这种方式限制人们的思维。从这个意义上讲，专用语言对程序员是有害的，尤其是那些在将来可能需要解决更大型问题的程序员。COBOL 语言就是一个很有说服力的例子。

对于 COBOL 语言设计者的最初意图，人们早已淡忘。这里不妨引用 Jean Sammet 的一段阐述：

“COBOL 的设计，实际上面向的是两类关心商业数据处理问题的用户：一类是那些经验相对欠缺的程序员，COBOL 的自然特性会对他们大有好处；而另一类用户则可能从来也没有写过程序。换言之，得益于 COBOL 程序的可读性，那些希望检查 COBOL 程序的人（无论是监督人员还是管理人员），都可以通过文档了解程序。反过来，几乎没有考虑到专业程序员的需要。……”

COBOL 的设计者费尽千辛万苦，才实现了他们预订的目标。他们努力的结果之一就是，在 COBOL 开发单位的程序开发主管，要比其他开发组的主管们读程序读得更多。但是除了程序开发主管之外，是否还有很多主管会阅读 COBOL 程序，或者其阅读效果是否好，就值得怀疑了。

60 年代，COBOL 在帮助非专业程序员编写程序方面的确取得了成功；但事实表明，正是由于这种成功，反过来却阻碍了 70 年代的进一步发展。对于一个计算中心来说，拥有一支 COBOL 程序员的开发队伍，并不足以去开发一些高级系统，比如在线的、需要快速反馈的系统。虽然后

来出现的随机存取的文件操作方式更有效，但是由于在使用 COBOL 过程中强制而成的顺序意识，这种技术的推广也受到了阻碍。在向我咨询过操作系统方面问题的单位中，75%的 COBOL 计算中心实际上都是把支持随机存取操作的文件当作顺序设备来使用。最好的也不过是可以更快地排序，而在最坏的情况下，顺序处理的速度要比用磁带还慢。无论是 COBOL 还是 COBOL 的用户，能否随着可以利用的条件的提高相应地有所发展，这是目前尚无定论的一个问题。

看起来这里有一个无法解决的矛盾——某种针对具体问题的特定用途，专门设计的语言，的确可以更好地完成这种任务；但是与此同时，它对其使用者思想的束缚也会更大，在他们以后需要解决新问题时，这种束缚就会暴露出来。实际上，这也是 Fisher 关于自然选择的基本定理的又一实例。如果是专业程序员，陷入这种表面看起来很适应，但实际上却适应能力很差的境地，就是不值得的；但问题是，我们无法在事先就知道，哪些人最终将成为专业的程序员。只要对专用语言对其特定环境的适应方式稍做研究，我们就很可能会理解，为了将业余或者半专业的程序员培养成专业程序员，在对其思维方式进行改造时必须考虑到哪些方面。

特定领域的用户可能也会用到一些特定的数据结构。而在对这些结构进行分析时，专用语言也可以大显身手——因为它们可以既实现简洁性，同时也保持与用户此前的经验联系起来。比如，在社会科学研究领域使用的某种语言，很有可能需要为处理调查问卷数据，预先提供数组数据的处理能力；而某种进行代数运算的语言，则很可能需要为多项式或者其他符号表达式的处理提供标准的数据格式。而实际上它们的确是这样做的：它们如果能够取得成功，原因正是在此；而要是它们失败了，往往是因为其结构的不合理。

我们也发现了与某些特定数据结构相关的一些特殊的子程序，它们可以完成对这些数据结构的常规处理功能。因此毫无疑问地，在面向社会科学研究的语言中，我们必须提供各式各样的数据简化原语；而在代数语言中，我们又必须提供支持解析式的乘法、微分运算以及化简表达式的各种原语。当然，在为这些函数和结构命名时，我们应该仔细考虑，以便让来

自不同领域的用户有似曾相识的感觉，从而可以充分利用他们在此前学习的成果。

在面向非程序员的专用语言设计中，对先前学习成果的利用是一个主要的目标，以免用户还没有来得及使用，就已经弃该语言而去了。不幸的是，许多发明——我实在不能用“开发”这个词——的专用语言，只提供了一种与用户前期知识的对照关系。虽然用户可以很快地掌握其使用方法，但是用不了多久他们就会发现，他们根本不能解决自己的实际问题，而只能“解决”那些与其实际问题相似的问题——也就是所谓的“玩具式”问题，尽管从理论上讲这种语言能够描述用户所遇到的实际问题，但是该语言系统与现实计算的内在配置极不协调，以至于无论是在时间还是空间方面，根本就不具备解决实际问题所必需的能力。一旦某位漫不经心的用户遇到过这种情况，他就很可能会成为一位强烈的批评者。而且，他的确可以理直气壮。

在当前的世界上，专用的程序语言不断被炮制出来（或许，这是因为某些人需要在自己的出版物清单中再添一笔）——专家们在阅读这些出版物时，会感到不知所云；而对于程序员们来说，这些根本就是多余的。但是，这种过于随意的做法，却不能不说是程序语言设计行业的耻辱。这并不是因为在多用途语言的衬托下，这种工作更引人注目；而是因为，如果脱离开计算机专业的核心部分，我们就会感到对社会更多的责任，因为其中的很多人不可能懂得如何保护自己，免受质量粗劣的程序语言的影响。在此，我们再次发出呼吁：不要再自以为是地去凭空杜撰出那些低劣的语言，还是让我们来营造一个具有合理行为准则的环境，并进而按照这些准则来进行语言的设计与测试。我们不仅要成为专业的程序员，而且要成为专业的设计师。

小结

Jean Sammet 写有一本名为《程序语言：发展历史及其基础》(Programming Language：History and Fundaments) 的书，在这本书封面上

的一幅画，令我们不禁想起 Brueghel[①] 的名画“巴别塔”。画的原作陈列于维也纳的 Kunsthistoriches 艺术博物馆[②]，但是在 IBM 的维也纳实验室（这里正是 PL/I 通用语言文档的诞生之地），环绕会议大厅有一幅此画的巨大复制品，占据了整面墙。在世界各地，凡是有程序员聚集的地方，大多都会有某样文字的或者图形的标志，其内容是关于通天塔的。只要看看 Sammet 那本书的封面，你就不难理解这是为什么——原来，这座塔的塔身是由 117 种程序语言的名字组成的。实际上，确切的数字应该是 118。尽管这本内容全面的书没有列出最后的这种语言，但是这种语言的确存在过，它的名字叫做 BABEL——就在塔尖上。

实际上据我所知，还有一些语言被遗漏了，甚至连一些“汇编”语言也没有被列出来。或许，Sammet 根本就没有打算在她的清单中列出这些语言，被列举出来的这 117 种语言具有代表性，它们前后跨越了 15 年时间，从 1952 年最初的 Short Code[③]，到截至 1967 年最新的语言。平均算起来，每年大概都有八种新的语言出现。值得一提的是，这种计算

① Pieter Brueghel（the Elder），（老）彼得·勃鲁盖尔（1525－1569）。出生于安特卫普东部的一个农民家庭，一生以农村生活作为艺术创作题材，人们称他为“农民的勃鲁盖尔”。他善于思想，性格幽默，喜爱夸张的艺术造型，因此人们又赠给他一个外号叫“滑稽的勃鲁盖尔”。他是尼德兰文艺复兴时期最后一位大师，他继承了凡·艾克兄弟、博斯等前辈的艺术精髓，以异常平凡、质朴、粗犷、豪放的笔调表现了日耳曼人、特别是日耳曼农民那种豪爽、乐观、憨厚、天真甚至有点滑稽可笑的人物形象。主要作品有农民的舞蹈（Peasant Dance，1568）、绞刑架下的舞蹈（Magpie on the Gallows，1568）、盲人的寓言（Parble of the Blind，1568）、伯利恒的户口调查（Numbering at Bethlehem，1566）、婚礼舞蹈（Wedding Dance，1566）、雪中猎人（Hunters in the Snow，1565）、行往守难地（Procession to Calvary，1564）、巴别塔（Tower of Bable，1563）、《叛逆天使的堕落》（Fall of the Rebel Angels，1562）、《荷兰人的寓言》（Nether landish Praverbs，1559）、《Dulle Griet》（1562）、《麻风病人》（Lepers，1568）、《死神的胜利》（Triumph of Death，1562）等。——译者注

② Kunsthistoriches 艺术博物馆，建于维也纳中心。1891 年开馆，收藏有哈布斯堡王族的大量艺术品。由绘画部、埃及部、古代美术部、雕塑工艺部、武器部及古乐器、货币和奖章部构成，还附设王宫宝库。其中的绘画部收藏最为重要，藏有 1500 幅绘画，包括乔尔乔涅、丢勒、勃鲁盖尔、荷尔拜因、鲁本斯以及样式主义的作品。其主页地址为 http：//www. khm. at/homeE/homeE. html。——译者注

③ 准确地讲，早在 1949 年，John W. Mauchly 就设计出了 Short Code 语言的最初版本；1950 年，该语言在 UNIVAC I 型机上得以实现和使用；而直到 1952 年，才由 A. B. Tonik 和 J. R. Logan 完成 Short Code 的修改版本。Short Code 被公认为是在电子计算机上实现的第一个高级程序语言。（但是，最早的高级语言可以追溯到 1945 年诞生的 Plan Calcul，它比第一台电子计算机的诞生还要早几个月；当然，真正得到推广使用、并且沿用至今的第一个高级语言，是由 Backus 设计、并于 1956 年首先在 IBM 公司的计算机上实现的 FORTRAN。）——译者注

方法可能会令你误解，因为在开始的 10 年时间内，每年出现的新语言的数量平均不超过两三种。而此后的 5 年内，这个数字高达每年接近 20 种，并且越是往后，这种趋势也愈演愈烈。虽然 Sammet 的这本宏篇巨著长达 785 页，但是或许再也不会出现此类著作——至少，有一卷的篇幅就足够了。如果我的推断没有错误，那么在 1972 年，程序语言的发明速度将达到每星期一种——如果把那些根本没有成文的也计算进来，那么这个数字还会更大；而要是把“方言”（同一语言在不同地区的变种）也计算进来，这个数字就难以估量了。

难道这类语言的泛滥，会像游泳池中水藻的滋生一样，糟糕透顶？抑或，这是我们对创造性成就的渴望的一种健康流露？或许我们曾经不费吹灰之力就可以“发明”一种语言，然后在文献中随意吹捧，或者胡乱地塞给在特定计算中心工作、使用特定计算机的程序员，但是这种日子已经一去不复返了。我们的视野应该更加开阔，同时要对那些自诩为发明家的人保持警惕——我们应该要求他们说明一下，从心理学的角度来看，他们“发明”的语言是否真的有用。不过，除了这些发明者之外，还有其他一些人也应该对此承担责任——比如程序语言的理论家们，很久以来他们一直都忽视了其工作在心理学方面的问题。程序开发并不是数学的一个分支，它是一种独特的交流形式，人在这个过程中扮演了一个积极的角色，而计算机则常常是消极的。或许，正是这种交流过程中的单向性，才导致了程序语言中的种种问题。只有在意识到程序“语言”中“相互交流”的一面，这些理论家们才会不得不承认：他们需要学习的，并不是对符号的操作，而应该是人的行为。

思考题

请主管回答：

1. 在你的单位中使用了哪些语言？你多久才察看一次程序？你是否觉得，程序语言对你的管理工作有所帮助？或者，你是否认为，程序语

言只是程序员们之间的一种“黑话”，通过这种语言，他们就可以背着你进行交流？

2. 在把一种新的语言引入到你所在的单位时，你会考虑到哪些心理的因素？你为此进行过何种实验？

请程序员回答：

1. 以你最拿手的一种程序开发语言为例，列举出其在下列方面的用途，并加以讨论：

(a) 一致性

(b) 不一致性

(c) 隐藏类型

2. 在你所知道的语言之中，哪一种是最为简洁的？它为什么可以如此简洁？它的简洁性是否取决于你利用它所解决的问题的限制？

3. 以你最熟悉的程序语言为例，列举出其可能引起的非局部性效应。在进行调试时，这种效应会产生什么影响？给出一些由于非局部性而造成程序错误的例子，并说明你最终是如何发现这些错误的。

4. 找来其他人写的一些程序，并根据其中同等长度的一段程序的可读性，将它们排个次序。然后计算一下其中的语句标志以及其中的分支指令或语句，并且检查一下，看看是否其中的分支越少，其可读性越高。同样地，计算一下每个分支的平均长度，并检查一下它与程序可读性之间的相关度。

5. 以你以前曾经使用过的语言为例，说明一下这些语言对你使用当前语言的影响及其方式。请举例说明，在程序开发之外的其他领域。你所掌握的语言是如何影响到你的思维过程的。

6. 如果你懂得某种专用的程序语言，就请描述一下，它是如何利用下列因素的：

(a) 问题涉及范围的限制
(b) 协调一致的数据结构
(c) 协调一致的处理函数
(d) 协调一致的术语

参考资料

- Rubey, Raymond J., et al., *Guide to PL/I, Vol. 1: Comparative evaluation* Detroit: American Data Processing, 1969

- Brooks, F. P., et al. *Guide to PL/I. Vol. 2: Experiences with PL/I Detroit*: American Data Processing, 1969

在评价新的程序语言方面，这两份文件针对 PL/I 语言，提出了两种评价方法。第二种方法的形式为传统一些的研讨班，在讨论过程中，参与者可以就 PL/I 在各个不同方面以及整体上的优劣发表自己的观点。但是第一种方法则不同，它是从行为学的角度来对程序语言进行研究，这类尝试为数不多。这种方法旨在消除不同个体之间的差异，为此它要求每位程序员分别使用两种不同语言编程解决同一个问题，使用的语言搭配方案可以是：COBOL-PL/I、FORTRAN-PL/I 或者 JOVIAL-PL/I。从事这项研究的人显然也是程序员，而不是行为科学家，但是他们并非技巧平庸之辈。如果你愿意，可以将这项研究与 Sackman 的研究进行对比，或许你可以得到介于二者之间的某种结论。

- Sammet. Jean E., *Programming Languages: History and Fundamentals*, Englewood Cliffs, N. J., Prentice-Hall, 1969.

除了其他方面的优点之外，这本书也堪称是一座关于专用语言及其设计的信息宝藏。

- Wirth, N., PL 360, A Programming Language for the 360 Computers, *Communications of the ACM*, 15, No. 1 (Jan. 1968), pp. 37－74

有很多人有一种强烈的感觉，认为只有某种汇编语言才可以称作为真正的通用语言。然而在注意到汇编语言暴露出来的种种弊端——尤其是其缺乏线性——之后，人们最终对它失去了信心。不过，Wirth 采用的方法为我们带来了一线希望的曙光。Wirth 的方法是，在将模块结构的优点强加于汇编语言的同时，仍然（至少在概念上）保持了对生成的机器级代码的完全控制。可以肯定地说，这是一篇最为重要的论文，它对于未来十年中系统程序的开发极具价值。

- Weinberg, G. M., *PL/I Programming: A Manual of Style*, New York, McGraw-

Hill, 1970.

请特别留意第二章，其中有关于在 PL/I 上下文中的语言设计原理的详细论述。

□ Klerer, M., and J. May, A User-Oriented Programming Language, *Computer Journal*, 8, No. 2 (July 1965). pp. 103－109.

从语言设计的角度来看，Klerer 和 May 的系统最令人着迷的地方是其对二义性的认可——实际上，用“欢迎”一次更为准确。其另外一个同样令人着迷的原因，是因为他们尚未给其语言命名——或者是由于谦虚，或许是自大。如果需要更加完整的评论，请参见 Sammet。

□ Shaw, C. J., *Decision Tables-An Annotated Bibliography*, Santa Monica, System Development Corp. (April 1965). TM－2288/000/000 (14 pp.).

□ Dixon, Paul, Decision Tables and Their Application, *Computers and Automation*, 13, (April 1964), pp. 14－19.

决策表运动（Decision Table Movement）在 60 年代早期盛极一时，后来一度转入地下，但是现在又卷土重来。不幸的是，该运动的复兴，似乎一方面只是侧重于形式问题，同时另一方面有关的书籍却在构思上欠考虑。为了从心理的意义上对决策表的价值作一评判，实验数据是必须的，但是我们仍然没有看到这样的数据。作者宣称，和基于过程的语言相比，他们介绍的方法更为自然——从这个意义上讲，缺少实验数据的确是一个遗憾。对于那些有志于在这片前途暗淡的领域中寻找希望的人们，我们为他们提供上述的参考书目以及综述文献，这些资料覆盖的时间范围一直到 1965 年左右。在本书写作的年代，似乎确实没有更好的参考了。

□ Weinberg, G. M., *Experiments in Problem Solving* (Doctoral Thesis), Ann Arbor, University Microfilms, 1965.

在允许自由选择心理模式时，被测者将会如何选择？本论文对此进行了研究；同时也连带讨论了在程序语言中最值得实现的结构类型。

□ Whorf, Benjamin Lee, *Language, Thought, and Reality*, Cambridge, Mass., M. I. T. Press, 1956.

Whorf 是一位人类学家、人类语言学家。通过对非“标准的美国－欧洲”语言的研究，他在语言对思维的影响方面有深刻的见解。这些见解的重要性可以与隐含类相提并论。其中一些针对美国印第安人语言的专门论述专业性很强，但是其他方面还是可以为语言学的门外汉们理解的。由于对其中论及的思维模式不甚熟悉，这些内容一般不易读懂。谁能说的准，也许技术性文章也会很有用处。有种族优越感的人也许会认为，只有欧洲的语言才可以为计算机程序设计提供有益的模型——但是，我们的确可以这样认为吗？或许我们会这么想过，但是在读完 Whorf 的书之后，你就再也不会了。



□ Zipf, G. K., *Human Behavior and the Principle of Least Effort*, Cambridge, Mass., Addison－Wesley, 1949.

Zipf 与 Whorf 一样，在智力方面都超出了其所处年代的范围，因此都经过了很长的时间才得到承认。如果你对诸如程序语言中的“方言”等现象有兴趣的话，那么 Zipf 关于语言变迁之原理的研究成果就值得你借鉴；尤其是对于那些对“无冗余”式程序开发有兴趣的人来说，这本书更是不可不读锁。

□ Shannon, C. E., Prediction and Entropy in Printed English, *Bell System Technical Journal*, Vol. 30 (1951), pp. 50－64.

关于这本书的讨论，请参见下一项。

□ Burton, N. G., and J. C. R. Licklider, Long－Range Constraints in the Statistical Structure of Printed English, *American Journal of Psychology*, Vol. 68 (1955), pp. 650－653.

这两篇论文都试图对印刷英文中的冗余程度进行测量，他们使用的方法也可以用以测量程序或者程序语言的冗余程度。但是这类实验却很难进行，比如在选择适当的程序和文本时就有困难；不过，我们在早期获得的结果还是显示出，不同语言之间确实存在这方面的差别。

□ Miller, George A., The Magical Number Seven, Plus－or－minus Two: Some Limits on Our Capacity for Processing Information, *Psychological Review*, 63, No. 2 (March 1956), pp. 81－97.

在这篇短小精悍而趣味盎然的文章中，Miller 阐述了人类处理信息的能力或多或少是如何限制其思维的，也介绍了我们的头脑采取了那些策略（比如组块）来克服这些限制。而这些策略似乎与程序员们使用的策略是一致的。

□ Smith, B., *Memory*, London, Allen and Unwin, Ltd., 1966.

□ Yates, F. A., *The Art of Memory*, Chicago, The University of Chicago Press, 1966.

□ Minsky, M., *Semantic Information Processing*, Cambridge, Mass., M. I. T. Press, 1968.

关于记忆力以及应该采用哪种类型的心理过程来设计程序语言，这 3 本书的观点各不相同。如果依照心理学的原理来设计程序语言，那么在各种心理过程之中会有大量的冲突，尤其是诸如记忆等“高级的”过程；而一个很自然的问题就是：哪些才是在程序设计过程中至关重要的？它们的确那么重要吗？

第12章　程序语言设计的一些原则

评　　注

在多年以前的一次采访中，有人问我："你认为自己对**计算机科学**的最大贡献是什么?"（其中的粗体字确实是当时的提问者问题的重点。）我当时的第一反应是，我对计算机科学根本没有任何贡献，而我也不认为所谓的"计算机科学"确实是一门科学。但是我还是在不经意间脱口而出："我对于计算的最大贡献就在于我从来没有发明过任何程序语言。"

虽然我不清楚当时为何会这样回答，但是我很清楚的是，那只是自己的一次胆怯的逃避。在重新回味最初的原话之后，我意识到那不过是为了使自己免遭别人的责难，而大多数程序员都会受到这种责难。为此，我不得不让自己处于一种"元"层次上。我确实没有设计过任何的程序语言，但是我也确实设计过一些原理，来指导程序语言的设计——也就是说，我是程序语言的"元"设计师。

作为一位元设计师，我就可以身处局外、隔岸观火，尽管妄加评论，而不必负任何的责任。所以今天我要坦白地承认：我曾经是最早的汇编语言的评论员之一；我评论过FORTRAN，我曾经与一位FORTRAN开发者共用过一台计算机，对其目标代码品头论足，还曾错误地预言说，用不了多久FORTRAN就会过时；我评论过APL，我曾经与众人一道，对Ken Iverson的工作吹毛求疵，当时他第一次提出自己关于语言的思想；我还评论过COBOL，这包括为宣传MetaCOBOL而作的努力；还有PL/I、Simula；还有IBM System/360的机器语言和任务控制语言——

在这方面做忏悔并不轻松；还有 Pascal——当时是在瑞士与 Niklaus Wirth 一起；还有用来形式地描述程序语言的维也纳定义语言[①]——当时是与属于 IBM 维也纳实验室的 Heinz Zemanek 的开发团队在一起。甚至对于 Algol（s）的早期版本、C 语言、Smalltalk 语言，还有由于我的工作（这是我最为得意的地方）而胎死腹中的其他数十种语言（其中至少有 3 种语言曾被冠以“神奇的”的美名），我也有很多可以大谈特谈的东西——尽管不知能起到什么作用。

在 1970 年赴东欧的一次旅行中，我惊讶地得知自己被尊称为“程序语言大王”。我对自己的认识就是这些。近些年来，我曾经尝试着忏悔自己的罪过、痛改前非，并退出对程序语言说三道四的行当，洗手不干，但是做起来实在太难了。而我所认识到东西，也许可以归结为程序开发心理学中的这样一条最为重要的原理：

每个程序员都是元语言的行家里手。

我想，对于名为“程序语言设计的一些原则”的一章，这是我敢于斗胆加上的惟一一句评注。

① Vienna definition language（VDL），维也纳定义语言。——译者注

第13章 其他的程序开发工具

尽管在已出版的文献中，程序语言赢得了最广泛的注意，但是从事实际工作的程序员还是会用到其他的一些工具。程序员需要花费大量的时间，来阅读或者撰写文档、与操作系统奋力搏斗或者绞尽脑汁地在程序中找出错误。在这些工作的过程中，社会科学家提供的哪怕是零星的工具，对程序员都是有用的。而从心理学的角度来看，与程序语言相比，人们要更加忽视这些工作。在接下来的各节中，我们将试图说明如何根据某些认识，为程序调试、程序开发上层的管理以及文档工作设计出更好的工具。我们不会太多地涉及到程序开发方面的资料，但是或许这种努力将开启设计模式的新纪元。我们肯定将使用这些方法！

程序测试工具

1962 年 9 月的一则新闻说：由于“在某一盘指令磁带中遗漏了一个连字符”，一支价值1 800万美元的火箭在开始的飞行中就报废了。报道中并未提及该“指令磁带”的尺寸有多大，但我们还是可以估计出，其中会包含100 000条左右的指令。在其他的任何行业中，100 000中出现一个错误将意味着很不寻常的可靠度；然而对于程序开发而言，则要付出1 800万美元的代价。这次付出的代价或许只是偶然的例外，但是这类问题每天都会不断地出现。如果仅仅从测试的角度来看，程序开发的工作确实是富有成效的——请注意，在程序开发中惟一的实质问题就是使程序工作正常，并且能够证明这一点。

有一句话说："所谓的专家，就是懂得回避小错误，从而找到大错误的人。"但是这句话并不适用于程序开发，因为任何工作中即使是忽略了很小的错误，都不能认为已完成了。而实际情况是，根本没有所谓的"小"错误——因为，即使是一个"连字符"的错误，也可能导致灾难性的后果。正是由于程序开发自身的特点，我们不可能估计出多么"大"的错误才会引发某些问题。因此，将程序测试（还远远不是"消除**所有的**错误"）的目标明确化是一件很困难的事——甚至可以说是一件无法完成的工作。

显然地，我们需要利用一切可以获得的帮助来进行程序测试。但是在程序开发工具的设计者眼中，程序测试的问题没有得到应有的重视。此外，即使是在设计测试工具时，人们也很难得会注意到程序测试过程中的心理问题。然而实际上，程序测试本身首先就是一个心理问题。比如，我们可以分析一下**信心**的问题。使用理想的测试工具会令我们产生信心，而且其产生的信心应该恰如其分——否则，我们就可能会放过某段仍有错误的程序，或者在已经没有错误时，还要浪费时间继续查找。在这个方面，当今的测试系统可以为我们提供哪些信息呢？

例如，如果要使人们相信对某段程序的测试结果，我们就需要了解，该测试在多大程度上覆盖了程序代码。我们在这里的意思是，在测试运行的过程中，测试系统首先应该时刻记录下代码中的哪些部分被执行过了、哪些还没有执行，并且动态地显示出来。在我们的社会中，人们都会自然地相信：事物会按照我们的意志而存在。因此在设计程序测试工具时，我们必须尽力使得这些工具可以真实地反映事物的本来面目。我们会很自然地理解到，要在测试工具把所有程序都执行过至少一遍之后，我们才有可能放心。

对于自己编写的代码，人们都倾向于持乐观的态度。因此为了设计出色的测试工具，就必须遵循"摧毁信心"的原则——当然，应该能够在必要时恢复这种信心。曾经有这样的一个程序，其功能是计算超过30 000个人造物品的价格。但是在4种情况下，它的计算有误。而有一次某位顾客恰巧遇到了这些情况，他对该程序的信心立刻因此丧失殆尽。程序员后来向他解释，只有当某个维度的长度超出范围之后，才会出现错误。但是，

这位顾客仍然对程序持怀疑态度。虽然在问题暴露时，只剩下一处出现错误的可能，但是由于这位顾客自己一下就遇到了 4 个错误，所以他就是再也不相信程序员了。

为了避免程序员的过分自信，可以使用一种系统，在测试程序时随机引入错误。错误出现的位置及其性质只保存在该系统之中，不对程序员公开。从某种意义上讲，根据程序员发现并且消除这些已知错误的比例，我们可以估计出他排除未知错误的比例。监视系统中还会例行地使用另一种类似的方法，让操作员目不转睛地监视雷达屏幕 8 个小时，而出现事件——比如出现身份不明的飞机的概率却很低。对操作员工作效率的测试结果表明，为了使他们能够保持足够的警觉，很有必要故意引入一些人为的事件，而且这种刺激的频率还不能很低。此外，因为这些事件都在系统的掌控之中，所以完全能够估算出每位操作员的当前以及整体的工作效率。

尽管我们无法效仿这种方法，在程序中人为地制造一些错误，来模拟真实的错误，但是在对程序员的程序测试技能进行培训与评估时，这种方法肯定还是有用的。即使需要让项目中的其他人为程序员的程序手工制造错误，但是很值得去尝试这类“制造错误”的方法。这种方法可以极大地提高程序员的积极性，因为这时他会意识到：我的程序中有错误，虽然不是我造成的，但是如果我不能找到它们，其他的人还是会知道。我们只需将这类事件的状态与目前的状态做一对比：如果不采用这种方法，程序员就不会真正承认自己的程序有错误；因为他知道就算发现了错误，那也是他“自己的”功劳；而要是他没有发现错误，别人也肯定不会知晓。

但是在对某个特定的程序进行测试时，程序员最终的信心并不是只取决于测试过程本身。为了树立起对自己程序的一定程度的信心，程序员到底需要进行多少次测试，在很大程度上还取决于通过演绎推理确定的程序正确的可能性。作为一个最简单的例子，让我们来考察下面的两段代码：

```
IF N = 1        THEN  A = A + 1.5;
IF N = 2        THEN  A = A + 2.9;
IF N = 3        THEN  A = A + 0.1;
...
```

```
IF N = 20      THEN      A = A - 3.2;
```

以及

```
A = A + B (N);
```

使用第二种形式，尽管我们在构造数组 B 的过程中也无法避免错误，但是显然出现错误的可能会更小。比如说，在使用第一种方式时，如果其中的某张卡片在穿孔时有误，以至于中间的某段可能会类似于下面的情况：

```
IF N = 11          THEN  A = A + 4.7;
IF N = 12          THEN  A = A + 2.6;
```

这时，N = 11 和 N = 12 的情况都是完全错误的，然而在测试中却可能不会出现这种错误。然而使用上述第二种方式后，各种情况之间就变成是相互独立的了——也就是说，对于任何一种情况，只要我们设置好了其在数组 B 中对应的元素，那么其他的元素就不可能对它有任何影响。此外，如果 N 比零小或者比 20 大，那么使用第二种形式就会产生一个 SUBSCRIPTRANGE 的错误条件，进而可以进行相应的处理；但是，第一种形式将忽略 N 越界的情况，而其后果将是灾难性的。

这些例子所说明的心理学问题是：在经过同样次数的测试之后，与第一种形式相比，程序员对第二种形式将更有信心。为什么呢？因为后一段程序具有一致性，而前一段程序却没有。只要对后一段程序做一次测试，就相当于对前一段程序做多次测试。这种一致性也许是从我们使用的程序语言得来的，也可能是通过程序结构的特定概念获得的，无论是哪种来源，其效果是相同的：在经过给定次数的测试之后，我们可以对程序更有信心。因此，可能的最优测试工具，必须同时建立在程序语言和程序结构的高度一致性之上。

通过类似的方法，我们可以证明：局部性或简洁性不佳的程序，将更加难以测试。但是，测试系统是否可以通过什么手段，对程序的一致性、局部性或者简洁性进行测量呢？这种方法并不存在，否则我们就可以在编译之后、开始测试之前，把那些杂乱的、散乱的或效率低的程序分拣出来，然后返还给相应的程序员及其管理人员重新审查。

我们没有任何的实验证据，可以说明这样的辅助测试工具是能够构造出来的。但是，我们还是可以为此方面研究的发展方向提出若干建议。首

先，可以通过对语句标号、GO TO 语句以及对定义于其他页面上的变量的引用进行计数，来衡量程序的局部性。还有更为复杂的衡量局部性的方法，比如计算出程序的某种“变量引用的平均距离”——这种方法对整个程序或者其中的片断都适用。在某些地方，还可能存在一些潜在的问题，比如，在半开放的子程序或者接通单元等处——因为它们有可能改变变量的数值。我们可以统计出这些潜在错误的数量，并且给出相应的警告；而在当前，编译器所给出的此类警告还远远不够。

一致性与简洁性的测量或许要更困难一些，但是对于特定的任务，如果能够在进行编码之前对程序的合理长度有所估计，那么就会有助于对程序的简洁性的评判。衡量一致性与简洁性的一个尺度，就是统计每个变量被引用的次数。如果对某个变量的引用次数很少，那就说明该变量没有必要设置，可以消除；反过来如果某个变量被引用的次数过多，那就说明可以改用一个不同的程序结构，以将分散的操作集中起来。我们可以统计一下，各数组有多少次不是通过数组表达式的形式被引用的，该统计值可以反映出程序一致性不佳的程度——因为从某种意义上讲，每个数组就是由同类的数据项构成的一个集合，所以应该按照同样的方式进行处理。

还有其他的方法可以发现程序在一致性和简洁性方面的缺陷，比如利用编译器来检测出代码中的特定“模式”。请考虑这样一个简单例子：在同一个程序中使用类似的变量名，比如 N1、N2、N3、N4 和 N5。正如我们知道的，首先存在的问题就是，在写或读的过程中，如此命名的一组变量很容易互相混淆，因此这种习惯会引发错误。不过，要是这些名称的背后确实包含了某种一致的含义，那么或许可以将它们代表的变量组织成一个数组，并进而按照一种更一致的方式处理。在程序模式检测方面，一个更为微妙的例子就是：雷同的一连串代码被反复使用。如果发现了这类重复现象，那就说明或许可以通过循环、子程序或宏定义等方法，把程序结构压缩得更紧凑；但是，有时这种现象根本不说明任何问题。对于那些在程序开发心理学方面初具概念与见解的人们来说，无论如何，这个领域都仍然有待深入研究。

人们在观察时会带有偏见，一种众所周知的这类偏见，就是由于过分依赖于“早期”数据结果而引起的。对程序测试来说，如果在程序开发过

程中很早就获得了“成功”，那么程序员就常常会过早地停止测试。为了防止这类错误，可以在测试过程之前就早早做好准备，如果有可能，最好在开始编码之前就着手准备测试。我们在这里所说的“测试”，是指狭义的“**检测**出程序中的错误”——而不是广义的任何测试。显然，我们无法在事先设计好测试，以**确定**某个错误的原因；即使错误已经发现，我们也无法在事先设计出**更正**错误的过程。尽管如此，由于任何测试都要从检测错误开始，所以在测试实例方面的前期工作绝对不是可有可无的——除非你被此前取得的“辉煌”成绩所迷惑，从而省略了后续的测试工作。

如果调试系统能够强制要求程序员事先计划好需要进行测试的工作量，那么就可以帮助我们克服上述麻痹心理。如果实际进行的测试达不到计划的数量，系统就会对程序员给出警示——比如生成一份管理报告。总而言之，如果测试系统能够简化测试样例的准备和运行过程，那么毫无疑问地，这将会有助于程序员克服盼望尽快结束测试的心理。如果只是对正确的程序进行了“很小的”修改，那么程序员往往不会再次进行测试；现在就有这样一个系统，可以帮助程序员抵制这种心理惰性。如果系统能够保存所有的测试样例，并且可以根据需要自动地重新运行，那么程序员就不大可能会忽略重测试的工作。然而，这类系统一般都会生成篇幅冗长的输出，所以程序员根本不愿意去查看这些输出结果。如果是这样，重新运行测试样例又会有什么用呢？这些系统的功能都需要改进，比如可以对测试结果的新、旧两个版本自动进行比较，这样就可以使程序员的注意力只集中于相邻两次测试的不同之处。

关于在程序测试过程中心理行为得到改善的可能性，我还有最后一点看法要谈。正如我们曾经在介绍“局部性”概念时所讨论的，如果程序员无法找到错误，那往往是由于他查找的位置有误。虽然道理再浅显不过了，但是一旦程序员的注意力被“锁定”到了某个错误的方向，要想让他改变关注的范围，谈何容易。一般而言，调试顾问可以向求助者提出的最佳建议就是，察看程序中别的地方——即使是在同样的地方，也要改变测试的方法。如果程序员对程序中的某个局部察看良久却没有结果，那么自动调试系统就可能会将程序员的注意力引向另一段不同的代码。虽然可以通过很多办法让程序员关注程序的不同部分，但是其诀窍却是相同的：尽

量把程序员从原来的思维惯性中解脱出来。实际上，这类系统可以采用一个再简单不过的方法：把终端屏幕暂时关闭一段时间，让程序员去看看树木、草地或者超短裙等，暂时忘却那段有问题的代码。

操作系统

关于程序语言的设计，有一个争论贯穿始终：调试工具到底应该放在何处——应该集成在语言本身之中，还是在操作系统中？在本节内只是摆出问题，虽然我们的讨论似乎有失偏颇，但实际上并没有给出任何结论。争论双方的理由都不够充分。一方面，程序员不必等到彻底掌握了某种语言，才开始着手进行调试——从人们对转储文件的憎恶之情，就可以明白地看出这一点。另一方面，如果只是停留在语言的层次上，程序员就不能察觉到操作环境中的某些实际应用因素——正是这些因素，才可能导致了程序测试过程中的最大问题。

至少从心理学的角度讲，两种策略正确的折衷结果似乎是：针对操作环境中的实际情况，将基本的程序语言在一定程度上进行扩展。但是这些扩展的引入，不能像操作系统的大多数功能一样，是杂乱无章的，而应该和语言的其他部分保持高度的一致性。然而，如果要在实现操作系统时满足一致性的要求，那么就要付出沉重的代价——因为通常的操作系统并不只是支持单一的某种语言，而是同时支持若干种语言。因此操作系统的设计者们如果能够为自己提供单独的一种语言，事情就会简单得多，而使用该操作系统支持的任何语言的程序员，都需要首先精通这种通用的语言。

以 OS/360 为例，该语言是基于 System/360 的汇编语言而设计的——我们已经对其后果有所认识。对于操作系统的设计者和实现者来说，这种选择似乎是天经地义的，因为他们觉得：难道还有谁不懂汇编语言吗？然而，即使是对那些掌握了汇编语言的人来说，汇编语言的格式也可能不是最适于任务控制语言的——至少，设计者们选择的那种格式确实不适宜。例如，OS/360 的任务控制语言就过分依赖于使用所谓的“定位参数”，而更少地使用“关键字参数”，但是，只有在用户能够熟练地运用该语言进行编码，并且希望将程序尽可能地压缩的情况下，定位参数才有可能发挥

最大的帮助作用。然而对于任务控制的通常用途而言，用户使用不同的选项不会获得多少经验，而且在生成自己的任务控制程序时，常常要在手边摆上一本参考手册。在这种情况下，如果使用安全而且确定的关键字参数标识，那么就不必记住次序，无需逗号来指示省略的项目，而且每个关键字都有一个助记的名字——因此，程序员会更喜欢这种方式。

在相对生疏的环境中进行程序开发，人们会在多大程度上倾向于使用关键字参数？我们的两位学生曾经就此做过试验。他们考察了由经验丰富或缺乏经验的程序员组成的小组，要求他们用某种形式为某些非程序问题进行编码，然后又使用其他的形式。他们发现，使用定位参数的编码进展得更快；即使是新问题，或者程序员缺乏经验，结果也是如此。但是在出现错误的比例方面，这种方式要高出 2～4 倍。考虑一下任务控制的情况，我们会发现：在这里，实际用于参数编码的时间很少，而且在查找这些地方的错误时，没有什么辅助的手段——因此，为了把编码速度提高 30%，所付出的错误量增加 3 倍的代价确实太大了。

在选定任务控制格式的时候，或许这个问题根本就没有被考虑过——因为人们似乎认为，这只是实现操作系统时所需要考虑的问题中很小的一部分。操作系统的设计者还会把另外一个问题也看成无关紧要的，这就是记账的问题——或者从更一般的意义上讲，应该称为性能评估。对于许多程序员来说，程序调试并非是最后一道工序。一个程序虽然已经可以运行了，但是它运行的代价可能过大。操作系统的设计者似乎很不情愿去了解一下：一个程序需要运行多长时间？它要占用多少内核？各文件中有多少记录需要处理？需要占用多少 CPU 时间？诸如此类。有些人甚至挖苦说：计算机制造商并不希望客户获得这类信息，否则他们就会发现，这种所谓“大型电子计算机”的实际工作能力是多么低。

无论计算机制造商的感觉如何，对于一个实现操作系统的任务而言，有关性能评估的信息毕竟是举足轻重的。首先，性能特性经常可以为排除程序错误提供不同的线索。如果一个程序运行过快，那么它很可能绕过了某些段的代码；而要是程序运行得过于缓慢，那么它就很可能是在不该有循环的地方进行了循环。然而除了对程序调试的这些益处之外，如果程序员准备学习能编写出性能良好的程序的方法，那么他们也会直接对此提出

要求。要是不能简单而自动地获得这种信息的反馈，那些最需要这些信息的程序员就永远也无法得到它们。此外，程序员可以对记账信息进行排序整理，其结果将有助于计算中心的管理人员搞清楚：哪些程序员需要帮助，哪些程序员可以提供帮助。只有每个计算机用户都要求制造商提供某种哪怕是最小规模的记账方案，我们才有可能取得计算方面的如此进步；否则的话，简直就是天方夜谭。

有时候，计算机制造商会辩解说：不同客户的需求互不相同，如果把要求的这类记账信息告诉他们，就会冒犯客户。无论如何，很难想像出比这更无耻的狡辩了，所以这更令人们相信那些愤世嫉俗者的讽刺挖苦。在大多数计算中心建立之初，我们当然要首先确定一些公共的需求，然后要为操作系统配备一套标准的记账程序。如果客户对这些程序不甚满意，他可以替换成自己的程序——虽然这样也会造成程序员的混乱，但是肯定不会比他现在更乱。实际上，由于操作系统还没有配备任何标准的记账程序，所以每当推出操作系统的一个新版本时，所谓“用户提供的”记账程序的界面总要相应改变——至少对于一家制造商确实如此。经过 5 到 6 个版本之后，大多数客户都会吸取教训，不再费力去紧跟新版本的变化。在相关的各类人中，似乎只有制造商才领会了心理学原理中最实用的一面——至少在这些原理可以用来回避那些他们不愿意做的工作时，确实如此。

如果我们列举一下一个记账程序应该提供的信息，那么首先一个方面就是：对批处理系统的平均轮换周期进行测量。当然，如果操作系统没有自行监督的远程信息入口和出口，那么记录完整的轮换周期就无从谈起；但是，如果不要求获得严格的完整数据，那么简单地统计一下计算机进入和退出之间的时间间隔，对解决大多数问题已经足够了。尽管如此，即使已经对影响轮换周期的各个因素有了完整的理解，我们还是会遇到同样（主要是心理方面）的问题：理想的轮换周期包括哪些部分？

我们的一位学生曾经着手研究这一关于理想轮换周期的问题。他在由程序员和主管组成的若干小组中进行了民意测验，方法是：投票者的个人信息被隐去，记录下其所属的小组。该学生的第 1 项重大发现是：不同小组对轮换周期需求的认识有所差异；但是在各小组内部，人们的理解相对

一致。推测起来，大概是因为不同性质的工作所需要的服务类型各自不同。他的第 2 项发现是：在有些小组中，关于小组的需求，主管与小组成员的理解竟然会截然相反。在其中一个小组中，一位主管经理正在致力于建立远程操作设备，以“改善轮换周期”；但是与此同时，小组的成员们却认为在自己的程序运行之后，他们需要更多的机会呆在机房里——因为，他们的工作是对操作系统中的输入/输出部分进行调试。

有些时候，主管会倾向于认为：程序员希望的是“零轮换周期”——其提交的计算任务总是立即就被接受，并且马上就可以获得计算结果。但是上面那位学生的调研以及其他一些观测的结果显示：主管们的这种理解与实际情况相左，程序员的要求并不是如此苛刻。为了对两种不同程序语言的功效进行比较，还进行过另外一项研究。在这项研究中，有一些参与者甚至抱怨说，轮换周期已经足够短了！原来，为了便于这项研究，这些程序员的优先级都被设置得更高了。于是统计结果显示，如果把轮换周期定义为从把计算任务通过服务窗口递交上去，到结果被送到放置计算输出的桌子上之间的时间间隔，那么其平均值仅为 31 分钟。抱怨者的理由是：如果轮换周期间隔再长一些，他们就可以有更充裕的时间来修正错误，并且通过对输出结果仔细分析，来发现更多的错误。

因此从某种意义上讲，如果程序员不用再苦苦等待即可得到运行结果，那么他对运行结果就不会很看重——在某些终端系统中，也存在类似的现象。相反的一种极端情况是：如果轮换周期极其漫长，那么程序员在提交计算任务时就会显得近乎神经过敏。在这种环境中，即使出现的只是一个不起眼的小错，需要付出的代价也是极其高昂的——程序员可能在漫漫等待中浪费一周的时间，最后却往往发现，自己不必要的神经过敏反而导致了一些错误。此外，程序员会尽量把所有能够想到的问题通过一次运行来解决，这同样会增加出错的可能，而且也会令轮换周期的状况更为恶化。在这类计算中心里常见的情况是，每个人出于自我保护的考虑都去提交耗时漫长的计算任务。这种做法无疑会使本已恶化的轮换周期状况雪上加霜。

因为不同计算任务的性质不尽相同，所以为轮换周期设置一个最佳的数值实际上是不可能的。通常，如果同一任务每天运行 4（或 5）次以上，

其生成的结果就会令程序员难以招架了；但是如果这个数字降到 2 以下，就会令程序员焦虑不安，而且其实际效果不佳。但是在工作开始之初则不然，因为在这个阶段，程序员的任务只是修正代码中的语法和输入错误，所以终端操作将是理想的工作方式。在那些每天有效运行次数很少的工作环境中，程序员会在这些“琐碎的”错误方面花费大量的时间；然而如果工作环境规定每天运行的任务数为 4 趟，那么程序员就显然不会在这些问题上遇到多少麻烦。

正如我们在其他一些语境中已经了解到的，**平均**时间并非惟一重要的心理指标。如果轮换周期非常不确定——有时 1 天 6 趟，但是第 2 天却 1 趟也没有——那么程序员就无法有效地规划自己的工作。为了应付这种不确定性，程序员必须能够很快地启动测试计算任务，以防范最坏的情况——这些只能导致精力的浪费，也会令程序员感觉极差。

优先权的分配方式，也是导致人们对轮换周期感到不满的另一个原因。在我们进行的一次调查中，程序员抱怨说，（其他的）某些人似乎不需任何明确的理由，就可以优先使用计算机。如果优先权的分配是根据工作的某种特性（比如预期的运算时间）统一进行的，那么就不会出现这种怨恨情绪；然而要是优先级的分配原则让人捉摸不定（比如没有原则或原则不一致），就会引起人们之间的敌对情绪，这种情绪既可能针对优先权的给予者（主管），也可能会针对优先权的获得者（程序员）。对一个程序开发组来说，任何形式的敌对情绪都极其有害，因此那些特殊需要统统不考虑也罢。

如果某种优先级制度授予某些类型工作的优先级比其他工作更高，那么制度实施之初就可能会受到人们的抵制，至少人们会感到焦虑不安；但是随着新制度的作用逐渐显现出来，这些问题通常都会慢慢消失。如果能够存储并公布一份良好的操作统计记录，将会有助于人们接受新制度，因为每位程序员都可以对各种操作的结果自行作出判断，并且对自己的程序运行做相应的调整。一种典型的策略是：需要运行更长时间的任务优先级更低，需要的核心存储容量超过一定范围的任务优先级更低，需要特别设置的任务优先级更低。最初，每个人似乎都会认为自己的任务应该是这些规则的一个特例；但是如果实际效果表明，大家都遵守这些规则会缩短整

体的轮换周期，那么人们渴望尽快提交各自计算任务的本能就会有所改变。运行时间与核心存储的预期状况会得到改善，那些曾经被人们极力标榜为“至关重要的”特别设置现在也似乎消失了。

上述情况，正是“效果律”在操作环境中的又一个应用实例。任何操作环境无论是好是坏，程序员们最终都将尽自己的最大努力去适应它。因此，即使程序员的行为有损于整体的操作，但是只要这种做法适用于他自己，而且他会因这种行为得到利益——或者至少不会因此而受到惩罚——那么所有的程序员最终都将采纳这种做法。比如在有的计算中心，要求程序员对自己的计算任务所需的时间进行估计，以帮助操作员更好地进行安排。但是正如一个真实例子中的情况一样，如果所有估计时间不超过一刻钟的计算任务都将被一视同仁，那么所有的程序员最终都会意识到，对他们个人而言最佳的一种方法就是，即使其计算任务只需 1 分钟，也要谎报为 15 分钟。

实际上，对程序将要消耗的资源，不必要求程序员做出超过操作系统反应精度的估计，这种要求毫无意义——因此在计算中心里，只需简单地要求程序员把自己的计算任务划分为“2 分钟以下”、“2 到 15 分钟”或者“15 分钟以上”，也许就可以同时指明各种选择的优先级别。类似地，如果是一个支持多程序的系统，而且计算任务只会按照大或小划分为两类，那么就只需要求程序员在二者之中作出选择——而不必精确地对任务需要的空间作出估计。如果作出规定而不严格执行，或者设定限制却不使用，那么这种做法只会招致那些必须在这些规定和限制下工作的人们的蔑视。

“效果律”的作用有时异常明显，但有时却极其微妙。此外，它也不只是对通常的“系统”起作用。只要对程序员的工作效绩有影响，那么无论是环境的哪一方面，不论系统主管是否认定它属于系统的一部分，程序员都将会加以适应。比如，Sackman 在他的一项研究中观察到，对于那些有 4 个以上问题的被试而言，其任务或者终端的等待时间都会下降很多。Sackman 是这样解释这一现象的：“随着学员们积累的经验更多，在申请计算机服务的时候，他们似乎会逐渐采用效率更高的时间策略。”

尽管 Sackman 把这一实验结果称为是“出乎预料的”，但是也许是因为他只是一位心理学家，而不是程序员，所以才会这样大惊小怪。任何职

业程序员都曾经会在自己或者其他人身上注意到这样的现象：无论是系统的何种特性，只要它对程序员的成功或者失败举足轻重，他们就会去适应这种特性。G. H. Stange 曾经对我们的程序开发心理学学习班做过 1 次调查，他调查的是两个远程任务提交系统的引入过程，而使用这两个系统的目的都是为了替换一个任务调度系统。他观察到了程序员适应任务调度程序的计划及其特性的很多例子，而其中一个最不寻常的观察结果是，计算任务的轮换周期居然是由其卡片打包的方式决定的！如果用橡皮筋捆绑卡片，效率是最好的（50%这类计算任务的返回时间不超过 3 个小时）；如果用硬纸盒来盛放卡片，效率次之（这类任务 3 个小时内返回的比例为 32%）；而如果卡片是用金属盒盛放的，效率最差（上述比例只有 12%）。Stange 怀疑在卡片容器与任务所需的时间或内核资源之间存在某种相关性，但是他的实验结果是否定的。很显然，操作员自然会优先考虑那些表面上**看起来**规模更小的任务——而且有些程序员已经开始意识到其中的差异。为了加以补偿，他们会竭尽所能地使其任务看起来规模尽量地小；而要是用金属容器来盛放一小叠卡片，后果将不堪设想。

在该开发中心引入第一个远程任务提交系统后，从理论上讲，这种对不同规模卡片的偏好应该可以消除，然而 Stange 的研究结果却表明，还有其他一些因素会导致程序受到区别对待，而且程序员会去适应它们。首先一个因素就是对新系统的了解，一方面人们会积极学习如何使用新系统，另一方面又不希望被别人察觉。然而随后人们就清楚地看到，未经训练的人们在使用系统的过程中，会造成卡片堵塞之类的故障，而这些故障只会影响其他人的工作。在这种情况下，更懂行的人就会来回巡视，担当起操作员的角色，以确保自己的计算任务不至于被延迟。在等待的过程中，除了确保系统不被搞乱之外，程序员只能无所事事——由于这种保护性的等待，新系统带来的许多明显的节约效果又都损失掉了。

程序员在终端前的聚集最终导致了其他一些效果——尤其是，使各自的程序在众人面前暴露得一览无余。打印机的速度极其慢，以至于聚集的人群可以看到每段程序被打印出来的详细过程，并且可以一边看，一边品头论足。与此同时，人们也会交换对远程任务提交系统（Romote Job Entry，RJE）本身以及程序开发的其他方面的看法。这样，人们的社会交

往就得到了加强；但是，由于很快就出现了一位非官方的“操作员”，使这种加强的过程从此中断，所以最终的效果不得而知。倘若系统时常崩溃，那么这位非正式的操作员就会站出来，保护其自身的利益——他会重新输入所有的计算任务，然而每次都会把自己的计算任务放在最前面。很显然，他之所以可以在提交计算任务方面享受到更好的服务，是作为他对系统进行维护而获得的报答；但是其他人都认可他的这种优先权，因为他们得到了好处，从此就不必在任务提交室内苦苦等待了。就这样，在经过短暂的混乱局面后，整个系统又重新回到一种稳定的状态；这时它的运转模式看起来非常类似于传统的督导员模式——也就是说，有一位督导员专门负责在程序员与“系统”之间进行调解。就这样，人们为了完成自己的计算任务，还是要借助人的能力，而不仅仅是操作系统。此外，传统的通过督导员进行管理的方法并没有被抛弃，因为这种督导员同时还可以为该领域的其他程序员提供服务。例如，在 RJE 系统运转不灵时，督导员提供的服务就可以作为备用的方案：如果计算任务需要使用大量卡片或者需要大量打印，也可以通过这种方式来运行，而无需对 RJE 系统进行任何调整；如果计算任务需要通宵运行，那么也可以采用这种方式——而在这种情况下，采用 RJE 模式却需要人进行看管。有一些程序员使用另一个 RJE 系统。该系统提供一个 CRT 设备，在对卡片组进行最初的处理时很有用；尤其是在需要对通过另外两种方式提交的计算任务进行监视的时候，就更派得上用场了。

综上所述，这两套系统的使用，最终都取得了预期的效果——每位程序员每天完成的计算任务的数量增加了，同时任务轮换周期缩短了。之所以会获得这种改进，问题能够得到解决，并非是由于简单地引入了新设备，而是得益于计算环境的改善——程序员可选择的余地更大了，在提交计算任务时，他们可以选择适合于自己的策略。当然，这一群程序员在获得更大的系统灵活性的同时，其他的程序员群体是否会因此受到负面的影响，我们无从知晓。最后，为了在不同的、相互竞争的用户之间保持一种平衡，必须建立起一套规则与限制。

然而，为了建立一套切合实际的规则和限制，操作系统必须要有足够的灵活性以接受和执行这些规则和限制。为了建立一套使用规程，也同样

需要操作系统的灵活性。为了保证系统的使用是高效的、有实效的，最好的办法莫过于使那些高效的、有实效的工作方式变得**最容易**使用。如果操作系统的设计良好，那么通过“列在一览表中的规程”，就可以更换操作系统的整体面目；这些规程用任务控制语言编写，并经常被调用，而且也应该经常被调用。通过这种方式，程序员们可以更加轻松地进行工作；这些好处可以对程序员产生引导作用，使他们逐渐学会如何更好地去使用计算机。

一方面，关于在不同计算中心里对记账程序的特殊需求，还存在一些争论；另一方面，某些制造商却——即使不总是在口头上，也是在事实上——极力撺掇用户去使用他们所谓的“标准”任务控制规程，而不是建立自己的控制规程。如果同时考察这两种现象，你就会感觉实在是很有意思。自然地，这些标准规程必须具有足够的一般性，才能够适应任何环境以及各种各样的命令，因此对于任何特定的工作或者环境而言，标准规程通常都不是优化的方案。也许，我们在这里所遇到的，是效率在更广范围内发生作用的一个例子。无论如何，计算机的使用时间越长，其制造商获得的利益也将越大。因此，只要是能够提高工作效率的东西（比如说记账系统），制造商就一定会竭力阻挠人们去使用；反过来，对那些将导致效率降低的东西（比如说泛泛的任务控制规程），他们就会极力怂恿人们去使用——这的确是再自然不过的事情。货物出门，概不退换；购者自慎，买主当心！

分时与批处理

计算机制造商有很多值得怀疑的方面，其中最大的一处莫过于他们对使用分时系统的反应。年长的人时常会听到一些人抱怨说，所谓的分时，只不过是为了进一步降低计算效率而玩弄的另一个伎俩，其目的只是为了往资本家们的口袋中塞进更多的钞票。可以肯定，分时系统的出现与计算机的其他革新一样，也是没有经过任何形式的心理学调研，就开始大范围地加以采用。人们认为这种方式行得通，或者乐于认定其行得通，所以就把它推向市场，并由此引发了竞争。

鼓吹分时系统的与提倡批处理系统的人们之间进行的激烈争论，大体而言曾经一直停留在相互谩骂的层次上。但是情况在突然间有所改变，人们在争论中开始引用试验结果，这使我们感觉耳目一新。这种变化得益于Sackman的著作“*Man-Computer Problem Solving: Experimental Evaluation of Time-Sharing and Batch Processing*”的出版。如果有的读者仍然对心理学方法在解决计算问题方面的能力持怀疑态度，就应该从头到尾地把Sackman的书通读一遍；在分时系统与批处理系统的论战中，鼓吹不同观点的双方也都应该去读读这本书。

Sackman对该问题的研究已经达到相当的深度，虽然在这里我们不指望能够触及这种深度，但是对于分时系统与批处理系统的论战给我们带来的教训，还是有必要做一讨论，因为在这个问题上，我们的经验最多。

有些假设会偷偷溜进某个领域，然后被大家当成是想当然的东西，即使是特别一丝不苟的研究者也可能如此——我们首先就要讨论这个问题。就Sackman而言，他把个人作为研究的基本单位——从来没有人对此有所怀疑，人们对这种做法决不会感到意外，因为这种假设正是所有分时系统的硬件、软件中的一个组成部分！比如说，没有任何终端适合两个或者多个程序员的协同工作，而且既然没有这种终端，也就根本不可能有支持这种终端的软件，甚至连这类软件的概念都没有。此外，由于Sackman的实验大多是在校园环境中进行的，所以他所使用的假设被进一步强化了。毫无疑问，在空军军官学校里，编写程序过程中的任何形式的合作，都会被认定是作弊——但愿情况不是如此！

我们可以从Sackman的工作中汲取的另一个教训是，即使是最细致的研究者，也有可能偶然犯错误，比如说错误地理解其试验结果。在他的一项研究中，Sackman给出了一张图表，用来说明在批处理状态下，语法错误的平均数目与批处理的运行次数之间的关系——也就是Sackman发现的“经典的错误消除[①]过程曲线”。尽管错误的确是被“消除”了，但是在心理学家的眼中，这根本就不能称之为“消除”。心理学家所说的“消除”是另外一回事——比如，被试为了按照要求记住一张列表，不断尝试并逐

① 作者认为Sackman偷换了心理学上Extinction（负强化）的概念。——译者注

渐减少记忆错误的过程。如果我们的研究与 Sackman 的小组有所联系，那么在批处理环境中工作的程序员，通常都要经历两个阶段。首先，他要尽力排除错误，以通过编译器的（语法）检查——这或许要经过 0 到 5 次运行。一旦完成了这一步工作，在通常情况下，他接下来的工作就是每次修改一条语句，通过这种方式逐渐逼近一个语义上正确的程序。虽然他可能会在某条语句上犯错误——如果他属于我们早先曾经定义过的“过于自信”类型，那么错误还会更多——但是这并不会受到影响到其程序的主体部分，程序员自己不需再做什么，程序已经是没有错误的了。实际上，错误是由穿孔卡片，而不是程序员的记忆来“消除”的。

既然错误能够被排除掉，那么至于到底是由谁来消除的，又会有什么区别呢？但是对于系统设计员来说，这种区别却是很大的。我给大家举个例子。几年前，曾经有一次我溜达到一家程序开发单位，当时那里的程序员正在为一个远程任务提交系统中的语法检查器确定技术规范。我对这个 PL/I 的检查器很感兴趣，所以察看了一下其技术规范。无论该检查器出自哪位程序员之手，都会被要求能够只根据一行上下文，就检查出“批处理编译器能够检查出来的任何错误”。试想一下，如果某位程序员企图在第一次编译时就成功通过，那么使用这个只根据一行上下文进行分析的语法检查器肯定是行不通的——因为，只根据一行上下文进行分析，无论如何也检查不出全局性的错误。为了确定这个检查器有多大的价值，我们就必须建立一套精确的错误模式作为它的输入。

根据我们的模型，可以这样解释 PL/I 程序员的工作过程。在首次编译成功通过之前，之所以要通过反复编译来消除语法错误，主要是因为没有正确地闭合注释或字符串。如果存在这类错误，（由此造成的编译中断）就会使大段大段的代码根本没有进行语法检查。然而，这——注释或字符串错误——恰恰就是批处理编译器无法检测到的两类错误，而且该语法检查器也将对这些错误视而不见。实际上，只要简单地对这两个问题进行检查，别的什么也不做，达到无语法错误状态所需的编译次数的平均值就将减少一次，而其方差值可能会降低得更多。

一旦成功通过了首次编译，剩下的语法错误大多都是那些原始的错误，比如说由于穿孔疏忽所造成的错误。这些错误往往是全局性的，比如

说某个名字在穿孔时出错了；有些错误更加琐碎，比如括号不匹配以及关键字拼写错误。在这种情况下，只有那种可以发现原始错误，并且能够根据此前的编译记录进行兼容性检查的检查器，才是最有用的。当然，如果每次修改都对程序动大手术，那么保存并维护符号表的做法就不值得了。因此只有明确地了解程序员的行为模型，我们才能知道应该如何去设计系统。如果我们像 Sackman 所认为的那样，每一次批处理程序的运行就其本质而言都是重新开始，程序员通过不断学习，来逐步“清除”语法错误，那么我们就会选用那种无需任何以往编译过程记录的方式，来进行语法检查。但是反过来，如果我们认为在解决一个问题的过程中，程序员关于语法的知识大体上不会有多少改变（尽管显然不可能完全没有），而且他们只是简单地以一次检查一条语句的方式来“排除”错误，那么我们对系统进行设计的方法就将非常不同——这种方法将令更优秀的程序员感到满意，并且给他们提供更加高效的系统性能。

关于系统性能问题的讨论，贯穿于 Sackman 工作的始终。尽管他和每一位优秀的科学家一样，尝试着把导致某种行为的成分分离出来，但是不同的因素总是相互纠缠在一起，相互影响，因此他的这种想法不可能实现。例如对于程序员们来说，“语言”与“系统”会互相混淆，批处理系统中的 ALGOL 和分时系统中的 BASIC 也会导致性能的差异。但是即使在两种系统中都使用“同一种”语言，它们还是有所区别，因为每个系统都有其特定的控制语句，这种语句在批处理与分时系统中是不尽相同的；另外，有的特性在某个环境中非常重要，但在另一个系统中却可能一文不值。

为了使批处理系统与分时系统“具有可比性”，还需要解决另外一个问题——这就是关于终端的可利用性与任务轮换周期之间的比较。假设某个（批处理）系统的任务轮换周期的平均值为 6 个小时，而其方差范围是上下 3 个小时；而在另外一个分时系统中，用户连续使用终端的时间被限定为最多 15 分钟，而在 90% 的时间内都可以申请获得终端，那么，二者之间具有可比性吗？Sackman 不断地遭遇到这类无法解决的问题，最后他得出一个结论：一个蹩脚的分时系统要比一个好的批处理系统更差；与此同时，一个好的分时系统却要比一个差劲的批处理系统更强。有的人可能

会认为，这个结论没有什么新鲜之处。但是从摒弃诸如“分时系统优于批处理系统”、“批处理系统优于分时系统”等空洞无物的陈述的角度来看，这一结论至少意味着一个巨大的进步。我们现在需要在 Sackman 的工作基础上搞清楚，是哪些因素使得批处理系统更具优势，又是哪些因素使得终端系统更具优势。不幸的是，恐怕我们将不能够按照自己所希望的那样简洁的形式，将这些“因素”分离出来——因为我们实际使用的系统并不像理想中的系统那样单纯。

最后，从每次任务运行中个体差异的角度来看，这种努力也许无论如何都是不值得的。或许，我们还是应该引用一下 Sackman 自己对整个问题的总结，以便我们了解关于批处理/分时系统的论战现在转向了何方：

> “或许通过这些研究所获得的核心结果就是，没有任何一个真实存在的例子可以说明，其中一种模式相对于另一种模式占有绝对优势。所有的研究都无一例外地表明，个体能力的差异要远远超出计算机系统本身之间的差异。为了在提高人机交互有效性的方面有重大突破，似乎应该从个体差异的特性与结构，以及人类的问题求解方式之中取寻求答案。一个机构的目标无论是为了更有效地传授计算机科学，还是为了开发出有成本合算的计算机程序，都需要通过一样的方法，才能有效地实现系统以及机构的目标——也就是要借助于人类在人机交流过程中的问题求解的基本知识。”

文档管理

文档管理只是程序开发过程中的蓖麻油[①]——主管们认为它会对程序员有益，而程序员却讨厌它！实际上，正是由于程序员如此厌恶文档，所以主管们才认定文档很有用——这就好比我们之所以会认定碘酒对我们膝盖上的伤口有好处，是因为我们能够感觉到刺痛。好吧，与在家中采用的民间医药难以根绝的情况一样，在程序开发的问题上，要想让人们相信文档管理工作与民间医药一样，没有任何内在价值，实在不是一件易事。只

① 蓖麻油可做工业机械（比如汽车引擎）的润滑剂，也可作为药用轻泻剂。——译者注

有在把文档管理工作做好之后，我们才能体会到文档的作用。相反地，如果文档管理工作做得一塌糊涂，那么就还不如干脆不要文档。

就我们可以肯定的部分而言，还没有任何方法能够**强行要求**程序员提供**良好**的文档。如果强行规定，有些程序员的确会提供一些文档——这就犹如如果强行要求，有些孩子确实会喝下蓖麻油；但是，如果程序员想要消极怠工，他们可以采用很多方法来生成毫无用处的文档，这样做只会导致一个结果——把大量金钱浪费在打印这类文档上面。所以，如果我们真的希望获得高质量的文档，惟一的方法就是去说服程序员，使他们确实相信做这些工作对他们自己有益处。如果编写文档的劳动对程序员没有任何价值，那么这方面的工作几乎就不可能完成；但是反过来，要是文档管理工作能显示出良好的效果，那么建立适当的高质量文档就**不会遇到任何阻碍**。

这里所谓的“适当的高质量文档”，具体指什么呢？这里强调的重点在“适当的”一词上，因为我们通常会犯的一种错误就是对文档管理工作给予过高的期望。首先，即使是在察看关于同一段程序的文档时，不同的人也会有各自不同的需求，因此奢望能够生成一份可以令所有人同样满意的文档，其实不过是天方夜谭式的幻想。我们设定的目标必须是合乎情理的，惟有如此才能根据这个目标，生成符合相应要求的高质量文档。要是碰巧这份文档也同时很好地满足别的目标，那不过是额外的收获；但是要没有这类巧合，我们将必须针对不同的目标编纂不同的文档，或者干脆不用任何文档。其次，文档并非万能的，有些任务仅凭文档是**不可能**完成的。这类任务的一个典型例子就是：只用不超过 25 个单词，就让一个未经训练的人了解一段程序的所有细节。

如果你想要从一段程序的文档中获得帮助，那就要求你做一些工作，并且要具备起码的知识。如果读者不懂英语，那么面对用英文写就的文档，他就会感觉像是在读天书一般；但是，该文档的作者却不会为此承担任何责任。但是同样地，如果某个人不懂 FORTRAN，而我们却要求他完全弄懂一段用 FORTRAN 写成的程序，那也是不近情理的。我们可以要求他掌握一段程序的**使用**方法，但其前提是：这段程序本身就是为了让人使用，而不是作为一个大系统的一个小部分而设计的——当然，他还要清

楚这段程序经过设计后的用途。然而，如果这个人在高中时的代数课程不及格，那么我们就绝不能奢望他能够理解一段求解微分方程的程序。当然最起码我们还是可以料到，一个受过高中教育的读者在看过文档的首页之后，应该能意识到如果继续阅读下去，自己能不能读懂。

关于文档，最重要的概念之一就是“深度”；对于大系统来说，尤其如此。如果系统的规模足够大，那么文档的不同读者在查找需要的信息细节时，需要触及到的深度也是不同的。在最浅的层次上，只要详细到能够让读者知道自己是否有能力阅读该文档就可以了。在每一个层次上——因为我们绝对不可能在事先就确定用户会使用哪份文档——文档中首先并且显著地标明，按照深度排列，它的上层以及下层分别是哪份文档；同时，还要指明该文档自己所处的深度，以及为了能够阅读该文档，读者必须具备的知识。

要是缺少这种起码的自我描述式结构，任何建立文档系统的工作都将是徒劳的；当然，读者也因此必须花费额外的时间，理解关于结构的信息，并且加以利用。我们不能指望，每次随手拿起一份文档，就恰好是可以帮助我们理解系统的那份——这种可能性，与我们直接从“*The Brothers Karamazov*”的第353页开始阅读，就能够理解Aloysha这个角色的概率相差无几。当然，在小说中会为读者提供一种标准的结构，而且读者对这种结构都是已知的，这极大地简化了作家的工作，他们用不着去确认，大多数认真的读者都会从第一页开始，一直读到最后一页。然而如果是要为一个大型系统编纂文档，文档作者们的写作能力本来就远远逊色于Dostoevski[①]或者James[②]，而且同时还要兼顾到有的读者不知道应该从哪里开始阅读。因此，每一本成形的文档都必须包括一定的指引信息。

当然，如果我们把自己对“成形文档”的概念只限定于书本的形式，那就不免有些过于短视。很有可能的情况是，只有在我们能够利用数据处理机器对自己的程序编写文档之后，计算机程序开发的文档管理问题才会真正得到解决——当然，这种自动化的形式必须同时具备创造性。可以肯定的是，现在我们已经开始朝着这个方向努力了——这就像我们通常向着

① Dostoevski，陀斯妥也夫斯基（1821—1881），俄国小说家。——译者注

② Henry James，亨利·詹姆斯（1843—1916），美国作家，评论家。——译者注

自动化处理的前进过程一样——在达到机械化之前，就将我们以往需要完成的任务机械化。这样，排版以及文字合成都可以通过所谓的“administrative terminal systems”[①] 来完成，而且在更新——这是程序开发过程中最为常见的问题——随时可以享受到自动化辅助带来的益处。然而基本上，因为我们还是要通过书本的媒介形式进行工作，所以仍然被限制于自己能够与文档读者进行交流的内容之中。

现在有的作者对继续使用书本抨击得有点过头，这种做法当然不甚明智。但是书本这种媒体所能发挥的作用的确有限。首先一个问题就是，从接受或者发布更新材料的角度来讲，书本并非是最灵活方便的介质。从被打印到纸上，到被印刷出来并第一次被阅读到，一般要经过漫长的时间——要是从最基本的原材料阶段算起，利用这段时间生个孩子绰绰有余。在有些时候，通过花更多的钱，或者牺牲内容的精确性——或者二者兼而有之——周转时间可以稍微减少一些；但是对于一个大型系统来说，由于其中的程序文档变更得非常频繁，以至于由于其本身的局限，书本介质难以跟上这种更新的速度。

第二个问题是，尽管有些人觉得坐在壁炉前，仔细地阅读一本书会令自己感到赏心悦目，但是无论如何，很少有人会在这样的环境中阅读程序文档。如果是一份笼统的概述性文档——比如，关于某个系统的一般性概念的，针对教学或者学习用途的文档——书本的确是适宜的介质。但如果是要回答一些特定的技术问题，那就还需要借助于一些其他的形式。一个复杂的系统必须要配备有大量的文档，但是置身于这样复杂的文档系统，我们不能保证用户不会迷失方向，因此我们必须建立一套主动式的指引系统，为读者们提供相应的帮助。利用计算机本身，通过适当的终端窗口，来实现这种主动式的引导——还有什么比这更好的方法呢？

嗯，不过这当然是有成本的。那么由谁来支付这笔费用呢？从心理的角度来分析，当被问及有关文档管理费用的问题时，人们总是会感到心头受到重重的一击，因为他们总是准备不够充分。原因何在？因为我们还没有习惯于把文档管理的费用，也作为程序开发费用中的一部分加以考虑。

① Administrative Terminal System (ATS)，行政管理终端系统。——译者注

因此当你从主管们的角度——金钱——来谈论问题时，你就会发现就其本质而言，他们同样认为文档管理工作是与程序开发独立的（或者说可以分割的）一个部分。拥有高质量的文档确实是很好的事情，但是如果需要为此付出金钱，事情就不那么妙了。

有的时候，通过某种形式的“自动化文档管理”，可以生成程序开发的规划方案——这些方案会被认为是唾手可得的或无需分文的，或者兼而有之。这类形式的一个例子，就是当前出现的大量流程图生成程序。你只要把自己的程序输入到这种程序中，它们就会给你输出一张流程图。这种过程肯定是——或者可以是——不需任何劳力的，而且实际上也看不出会有多么昂贵。但是，难道这就是所谓的文档管理？也许不是。充其量而言，这不过是程序调试工具库中的一个工具罢了——不过，这个工具也不可小视。

作为一名程序员，如果在工作的时候没有自己的流程图，就会对很多事情搞不清楚，甚至连自己编写的程序是干什么的也不了解；而要是程序的局部性不好，问题就显得更加突出了。反过来，要是有一份准确的流程图在手边，那么就可以按照二维表格方式，把许多非局部的信息流节点汇总起来，从而可以在区区一页纸上一览无余。这种关于程序的新理解，对程序员来说极有帮助；它不仅仅可以帮助查出一些固定性错误（比如未能闭合的循环，或者没有链接到的分支），而且可以显示出这段代码是否真的实现了作者的本来用意。

在工作中，一名程序员若是自始至终地都使用流程图，那么计算机绘制的流程图可以用来进行比较。在把流程图转换为具体代码的过程中，程序员可能会犯错误，通过这种比较，肯定能够帮助程序员找出这些错误。不过在线性程度更高的语言（比如 PL/I）中，流程图的二维结构所具有的优点就显现不出来了。如果把一段没有标号的 PL/I 程序输入到这种流程图生成程序中，输出的将还是原来的代码，而且顺序也不会改变，但是在不同的代码片段周围，会画上精美的小方框。尽管流程图生成器在代码的周围加上方框可以帮助程序员发现一些原来没有注意的问题，但是有些任务它却不能完成，比如说在方框**内部**添上点新的信息。

然而，倘若除了程序的原作者之外，其他人都看不懂方框中的信息，那么根据它们自动生成的流程图——无论是用何种语言——对于后来需要

文档的人来说，都不会有多大的帮助。如果原来的注释就令人捉摸不透，那么新的注释也同样让读者感觉一头雾水——除非只是把分支指向的目标弄错了。纵然符号选择不当，它们在流程图中还是会被沿用下来。为了测试这种自动生成的流程图作为文档的价值，我们进行了一些初步的研究。我们没有发现任何证据可以证明，借助这种流程图来阅读原始代码，会比直接阅读原始代码轻松多少——当然，进行最初编码的程序员要除外。贩卖这类流程图生成程序的公司，也许应该有几家愿意为针对这个问题的更精细的研究提供资助。

当然，一份流程图只不过是表示实际代码中的信息的另外一种方式罢了。尽管其总体的价值的确不高，但是作为一种文档化的技术方法，它还是有其存在的原因的。首先，由于这是一种由计算机自动完成的过程，所以我们有理由相信，随着计算机的能力不断增加，其成本会降低，而不可能增加。其次，它可以根据代码直接生成，而这绝对是一项突出的优点，因此维护文档的工作与维护代码的工作就完全成为一码事。因此，根本不会出现所有文档系统都存在的致命问题——时间滞后。

文档应该能够根据代码本身快速地生成，Klerer 和 May 关于他们的系统的评价，是这种观点的一个极端的例子。他们为自己的系统编写的文档，只需要一张纸的正反两面就可以写得下。不仅如此，他们还明确地表示，他们并不指望程序员通过他们提供的文档就能够掌握如何使用该系统。他们认为，只有通过使用系统本身过程中的经验积累，才能真正地掌握它。有的文档不惜使用成千上万个字来说明某个问题，但是这个问题过于晦涩，以至于程序员十有八九根本不会去触及这个问题。与这种做法相反，Klerer 和 May 把清理程序中不明确之处的任务交给计算机自己来完成。也就是说，归根到底我们只不过是又重新回到了我们多年以来一直沿用的方法上来。如果存在关于文档管理的争论，那么就需要计算机作出最后的裁决；而且，情况将永远如此。

但是在近些年来，计算机却成为了一种少言寡语的文档生成器。为了能够提高计算机的可依赖性，我们必须更多地运用想像力，使得计算机可以用尽可能多的手段来提供信息。每一种特定的手段（比如流程图生成器），只能在很小的程度上提高程序的清晰性；但是很多个小小的提高积

累起来，其效果要远远超出其中任何一个。如果真像我们所认为的那样，程序调试与理解过程中的主要障碍来源于心理集合，那么无论是何种技术方法，只要能够帮助人们从心理集合中解脱出来，它就有可能成为调试和文档化的工具。

阅读程序的其他方法，效率又如何呢？我们对此进行了一系列的研究。我们搜肠刮肚、绞尽脑汁，期望能够提出不同的构想。比如说画家们会有这种体会，即如果对物体过于熟悉，那么就可能使他对物体的真实色彩的感觉产生偏差。为了克服这种心理集合，他会俯下身子，然后从上往下，从两腿之间去观察风景。仿效这种方法，我们做了类似的实验，即按照相反的顺序——以字母或者符号为单位——把程序打印出来，为的是看看会有什么效果。

我们正在尝试的其他方法中还有一些是大家比较熟悉的，不过我还是想在这里把它们罗列出来，希望能够起到抛砖引玉的效果：

1. 把关键字设置为粗体或者加上下划线；

2. 把非关键字的内容设置为粗体或者加上下划线；

3. 把关键字或者非关键字设置为小写；

4. 更换具有多种用途的符号（比如说具有二义性的等号，以及在不同级别上相互匹配的括号）；

5. 把可能存在问题的位置标记出来（比如说有些地方生成了哑元，但是实际上应该是引用；或者有的地方虽然使用的是引用，但是实际上需要的却是哑元）；

6. 把程序清单中的注释剥离出来，把纯粹的程序集中到一页纸上，而把注释集中到另外一张纸上；

7. 使用上述方法，处理常量[①]；

8. 将所有变量重新命名；

9. 把所有缩写之处都展开，或者把所有的展开部分都缩写；

10. 将选项按照某种标准顺序进行排列，或者简单换个次序；

11. 按照字母顺序，将所有符号列举出来，同时给出相应的引用指示，

① Literal，源程序中的一个符号或量，它本身就是数据，而不是数据的引用。——译者注

以便于查找它们是在什么地方被用到、被设置或者作为引用被传递；

12. 根据某些特定的属性，将所有的符号分组——比方说，所有的EXTERNAL（外部）变量或者所有的 FLOAT 类型变量集中到一起，同时根据多重属性建立多重列表；

13. 列出所有符号作用域的清单；

14. 列举出潜在的二义性；

15. 对于 DO、BEGIN 以及 PROCEDURE 等匹配结构，按照其连续的嵌套级别，将它们抽取并打印成独立的群组；

16. 将条件语句分解成某种正则形式，也可以包括决策表；

17. 把表达式剥离出来，对其中包含的变量作几次赋值，然后分别计算出其数值，最后把这些数值罗列出来；

18. 设计出程序流程的几个例子，然后实际执行。

这些观点中，没有任何一个试图去改变程序的物理意义；但是，它们却都试图去改变程序的心理含义。我们的用意在于，使程序员可以获得多方面的观测结果，他们不仅可以用眼睛（或者通过其他感官）看到这些结果，也可以用大脑去进行相应的思考——这样，他们就可以犹如一台自适应的机器一样，将自己原有的天赋发挥得淋漓尽致。换而言之，我们并非是要利用计算机来解决程序调试以及文档管理的问题；我们的目的在于，利用计算机来帮助人，使他们能够利用自己巨大的心理资源，来克服巨大的心理弱点。

倘若这些程序**的确**能够解决（或者部分解决）文档管理的问题，那自然是再好不过的了。在这方面，还有另外一种方法值得努力去做，这需要程序员完成些许额外的工作，但是同时也看起来很有用。根据这种方法，程序员一旦写完了程序，他就要在一台终端前坐下来，然后接受计算机的盘问，他会被问及代码中不同程序段以及数据元素的功能。他所做出的回答都会被保存下来，接下来如果有人需要该程序的一个拷贝，他一般就会获得这样的一份记录。然后，用户也可以找到一台终端，并且提出一些自己感兴趣的、有关程序以及数据元素的问题——由于这时有了计算机的帮助，所以他们就可以不必再到浩如烟海的文档之中去查找了。查询信息时他们再也不会找错地方了，因为每次都是去同一个地方——终端。

在那些不甚了解文档结构的用户看来，对于他们所面临的问题，这类系统似乎是了如指掌；然而，他们的问题并没有得到解决，文档数据的输入方式还是照旧。与计算机的交互过程，毫无疑问地会提高一些程序员积极性；不过，纵然是最有积极性的程序员，倘若他连完整的英语句子也写不出来，那么就绝对不可能通过使用这种系统而成为一名称职的交流者。有一种变通的方法，也就是仅仅把计算机当作一名裁判，在程序员与若干“一般的”用户进行探讨时，负责进行评判。用户可以通过其终端提出问题，而程序员则可以试着通过自己的终端回答这些问题。计算机甚至还可以给用户一些提示，以帮助他们提出某些问题；但是，只有等到程序员的回答被用户认可为对自己的确有用之后，这些答案才会被记录下来。所有有用的回答都会被分类整理，然后随同程序一起发送给其他用户，继续进行这类询问。

在可以预见的未来，可能会有一个能够集中回答这些问题的数据库——当然，这种方法会更好。系统的用户（或者潜在的用户）将能够向这个数据库直接进行询问。尽管可以通过统计确定最经常问到的问题，从而可以在将来更加直接地给出这些问题的答案，但是在计算机的指导下，只要用户能够获得其所需的信息，也就可以了。不过，要是用户对于数据库中目前能够提供的回答不甚满意，计算机就会在计算中心里找出某个人，请他与该用户直接联系。在他们经过交谈之后，问题最终总会解决——当然，在最坏的情况下，这可能要等上好些天。一旦找到了答案，它就会被添加到记录当中，于是以后的用户就可以使用了。

这种系统所做的工作，不过是提供了一种适宜的机制，使在发生变化时可以进行更新；通过这种方式，它将我们带回到最初的时间，那时还没有任何文档——如果要说有，那就是程序员自己。如果是在小型的开发单位，开发的是小型的程序，那么人们就不会过多地注意显式的文档，因为在这种情况下，很容易就可以找到最初的程序作者，并且从那里得到解答。无论如何，你也有可能必须向他进行咨询，以找到想要的文档。在当前的条件下，这实际上或许就是最佳的办法。对于自己编写的程序，程序员总是最了解的——这种了解的程度，要超出他能够在文档中通过注释记录下来的信息；即使他的确记录下来了，而且用户也能够找到相应的记

录，用户也不会理解到原作者的层次。这种方法也不是尽善尽美的，比如说，一个主要的问题就是程序员有可能会离开开发单位；但是，只要采用无私式程序开发方法——而且那里的管理工作不至于很糟糕，以至于所有的程序员都同时离开——那么这个问题的负面影响就会降至最低水平。

很自然地，这种个性化的方法也存在局限。在 1961 年出现过一个特别奇怪的情况，它涉及到我于 1956 年在旧金山工作时编写的一个网络分析程序。在那些年中，这个程序是非正式发行的，而且得到了广泛的使用。但是在那几年之间，我始终都在从这里游荡到那里，直到最后才在纽约落脚。有一天，我接到一个来自斯德哥尔摩的长途电话。打来电话的是一位焦急的程序员，他整整找了我 3 天。他解释说，自己正在试图修改这个程序，以使它可以处理该问题最初定义中没有提到的某个情况。他问我："要是我将1 003处的符号改成负号，是不是就可以实现这种功能呢?"

由于在那 5 年的时间内，我一直往返奔波于各大洲之间，所以我当时很难记起自己曾经写过这个程序，更不必说记起 1 003 处到底是什么内容了。我当时飞快地心算出这种跨越大西洋的电话费用，以及计算机机时的费用，经过比较之后我对来电话者说："是的，确实可以实现。但是，你还是应该上机实际尝试一下。如果有问题，可以再给我来电话。"幸运的是，这位瑞典朋友后来再也没有来过电话。

虽然从外表上看，程序员没有什么变化，但是其记忆的东西却会渐渐地被淡忘掉。很少会有程序员能够在写完某个程序之后，经过 5 年都没有再去看过它，却依然能够回忆起在任何指定位置处的内容是什么——无论是外在的符号，还是内在的含义，这都是不可能的。但是如果这个程序本来就是只供一个人使用的，那么就不要指望经过 5 年的时间，这个程序依然在被使用。无论其文档有多么好，只要它的寿命达到这样长之后，就应该被彻底丢弃，从头再来。

这样，在确定如何对某个程序作文档时，我们所要考虑的应该是"谁将会使用到这些文档?"、"使用方式如何?"、"用户来自哪些地方?"以及"文档将会被使用多长时间?"如果只会在局部用到，只涉及到按照设计需要处理的工作，使用文档的用户对程序员很熟悉，而且只会被用到一两年或者至多 3 年，那么花费在整理文档上的金钱或许应该花在其他一些项目

上更好。如果有很多的远程用户要使用该文档，那么制作文档的成本就必须从工作之处开始计算；为了保证文档化工作可能成功，程序员从一开始起就必须参与到此项工作中来。

需要提醒读者的是，我们在这里说的并不是“文档化”，而是“文档化的可能性”。如果某个程序成为了销路很好的商品，那么无论实际上它是否真的卖出去了，用户都有权利要求文档的制作能够像程序开发一样达到专业水准。只要冷静地考虑一下这个问题，我们就没有理由相信，一名职业的程序员同时也能做一名称职的职业文档员。所以，如果需要制作出专业水准的文档，就应该在程序员的**协助**下，而不是由程序员**自己**来完成这项工作。一旦文档本身被提高到了专业的水准，文档员就可以与程序员并肩工作，而不会感觉自己低人一等，那么我们就有理由相信，文档的质量将会得到提高。

从创造性地将程序开发本身看作人的一种行为那一刻起，我们已经走出了程序开发的整个怪圈。那些对程序开发似懂非懂的行政人员错误地认为，任何人都能够编写程序；而程序员们则努力使人们承认他们是真正的专业人员，如果把这种努力比做一场战争，那么直到今天他们才终于获得了胜利。并非任何人——无论其背景如何，无论受过任何训练——都能够胜任程序开发这种工作的。然而，既然程序员懂得这个道理，那么他们为什么却又认为随便到大街上找个人就能胜任编写文档的任务呢？只要花上个把小时看看毕业生写的论文，你就会明白相互沟通能力之缺乏已经到了何种地步。因此，在讨论计算机程序文档的问题时，我们根本就没有涉及到计算机程序开发心理学——只有程序员们的一种误解是个例外，他们总是错误地认为，虽然不一定每个人都能够成为程序员，但是任何人都可以毫不费力地胜任制作文档的工作。

小结

我们可以从本章获得的认识很简单：系统是复杂的。任何计算机系统绝不仅仅只包括硬件，也不仅仅是软件，甚至于人加上硬件再加上软

件也还不够。无论是正规的还是非正规的，任何涉及到系统的过程都是该系统的一部分；不仅不同部分当前的负载是如此，而且用户的态度与经验也是如此。即使是按照广为接受的定义，系统各“部分”之间也没有明确的分界线。首先是硬件与操作系统互相融合，操作系统进而又与程序语言互相融合，程序语言再与调试工具互相融合，调试工具再与文档化互相融合，文档化再与培训互相融合，最后，所有这些方面都会与系统所处的社会环境互相融合。

这种复杂性一方面意味着试验结果的加倍重要，同时另一方面也意味着，要获得和解释试验结果也将加倍困难。现在是把这些无稽之谈从我们的头脑中清除出去的时候了，如果可能，我们应该代之以实在的实验证据。重要的并不是其中哪个问题将会被“解决”——因为其复杂性实在是高得难以克服；而关键在于，哪些有价值的问题能够得到“解决”？无论如何，我们关心的并非是获得了多少答案，而是在尝试寻求答案过程中的经验。

思考题

请主管回答：

1. 是否曾经有人向你推销过某种“调试编译器”或者其他的调试辅助工具？你是根据什么来决定是否购买它的？

2. 你如何要求手下的程序员对自己的“调试”程序工作的自信进行估计？你是否尝试过自行对信心进行估计？在进行这类判断时，你根据的是哪些因素？

3. 请介绍一下你所在开发单位对工作效绩进行评估的体系。作为一名主管经理，你如何应用这些体系？你还希望通过这个体系获得哪些其他的信息？这个体系所提供的信息中，哪些是对你毫无用处的？

4. 在自己所在的批处理系统中，你认为什么样的轮换周期才是理想

的？在你所在的分时系统中，你认为什么样的操作方式才是理想的？关于这些问题，你手下的程序员是如何想的？

5. 在使用批处理系统时，你是否给了某位程序员优先特权？在使用分时系统呢？关于你的这种偏好，你是否注意到过有什么不利效果？为了在授予某个人优先特权时不至于使其他人感觉不好，你认为自己在将来会怎样做？虽然某些人的优先特权会带来好处，但同时该其他人带来的不好感觉却会产生更坏的效果——你是否认为有这样的可能性？

6. 对于程序开发而言，分时系统与批处理系统哪种要更好些？

7. 在你所在的开发单位，程序文档员是如何挑选出来的？你们的文档制度是否足够灵活，从而可以为不同类型的程序建立不同类型的文档？是否存在对文档建立工作的不满情绪？如果有这种情绪，你认为其根源何在？应该如何去解决这个问题？

请程序员回答：

1. 请回忆一下你的经历，是否曾经因为某个“微小的”错误，最终付出了巨大的代价？为了防止这种错误，当时应该使用哪些调试工具或者技术？这些工具或技术的成本是否比那次错误的代价还要大？

2. 试举例说明，你曾经因为程序的“快速返回”，从而对自己的程序测试产生了（好的以及坏的）片面理解。举出其他一些例子，说明还有哪些因素使你对调试过程中使用的策略产生了理解的偏差（比如说，渴望取悦你的主管、对程序结构的了解以及与其他程序或程序员的交流）。

3. 在自己所在的批处理系统中，你认为什么样的轮换周期才是理想的？在你所使用的分时系统中，你认为什么样的操作方式才是理想的？关于这些问题，你的主管是如何想的？

4. 试举出你自身的例子，以说明在行为方面对系统特性的适应过程。

5. 对于程序开发而言，分时系统与批处理系统哪种要更好些？

6. 你希望自己所使用的操作系统和编译器还应该提供哪些分析程序的形式？对每一种形式，请解释该种形式会带来哪些帮助，并给出具体的例子。

7. 你对建立文档的工作怎么看？对那些能够出色完成文档工作的人，你如何看待他们？为了帮助你圆满地完成文档建立工作，你认为哪些方面应该有所变化？

参考资料

□ Rosenthal, Robert, On the Social Psychology of the Psychological Experiment, *American Scientist*, June 1963.

关于研究实验过程中混杂的观察性偏见，尤其是关于由于实验者的期望而导致的偏见，Rosenthal 是这方面的先驱。程序调试与之有明显的相似之处，所以程序员应该能够将这篇文章的结果运用到自己的工作实践中。

□ Parzen, Emanuel, *Modern Probability Theory and Its Applications*, New York, Wiley, 1960.

对于某一特定测试的结果，我们的相信程度会受到先验概率的影响；关于这个问题，请参阅第 119 - 120 页。一般而言，如果程序员能够把程序调试看作一个概率过程，就会受益匪浅，而 Parzen 的这本书就是很好的启蒙读物。

□ Weinberg, G. M., *PL/I Programming ——A Manual of Style*, New York, McGraw-Hill, 1970.

请注意第 4 章——这里讨论了在程序测试过程中，使用恰当的语言和正确的程序结构所带来的益处的讨论。

□ Berkeley, Edmund C., The Personality of the Interactive Programmed Computer, *Computers and Automation*, Dec. 1965, pp. 42 - 46.

虽然这篇文章略显肤浅，但是它却提出了一个有趣的问题，即关于系统对用户呈现出的“人格”问题。正如 Sackman 已经说明过的，一个用户对系统所持的态度会反过来影响其使用该系统的工作效绩；同时，随着用户对系统日益熟悉，他们对系统的认同程度也会与日俱增。在这样一个逐步接纳的过程中，系统所具有的“人格”毫无疑问也起了作用；同时为了确定程序员到底愿意使用具有何种人格的系统，可以对用户关于不同系统所具人格的理解进行测试。

□ Sackman, Harold, *Man-Computer Problem Solving: Experimental Evaluation of Time-Sharing and Batch Processing*, Princeton, N. J., Auerbach Publishers, 1970.

正如其副标题所指示的，同时也是我们早先曾经指出过的，这本书将该领域的工作按照年代进行了总结。因此，对于那些准备对操作系统进行比较的人来说，这是一本重要的必读书。

□ Stange, G. H., The Initial Effects of the Installation of Two Remote Computer Access Systems.

未出版的内部报告。

□ Salsbury, R. G., Ideal Turnaround——An Opinion Survey.

未出版的内部报告。

□ Heiss, William J., and Frederick Schwartz, Positional versus Keywood Coding.

所有的有关人员都可以从这些未出版的内部报告中获得有关学习的经验，这些报告同时也说明，在这一领域做出有用的工作，并不一定需要深入到最底层。

□ Rosin, Robert F., Supervisory and Monitor Systems, *Computing Surveys*, 1, No. 1 (March 1969).

虽然这份调研报告只有短短的十几页，但是 Rosin 还是成功地阐述清楚了有关操作系统的全面要点以及重要的问题。透过历史的角度进行观察，他成功地揭示了在代代相接的操作系统之间，其设计过程或多或少地会倾向于去修补前一代操作系统所遗留的问题。读完这份报告，人们或许会猜测，如果朝着修补当前操作系统在心理方面存在缺陷的方向发展，未来的操作系统将会是什么样子。

□ London, Ralph L., Bibliography on Proving the Correctness of Computer Programs, *Machine Intelligence*, 5, New York, American Elsevier, 1970.

所有的程序调试员都有这样一个梦想：有朝一日造出一台机器，它能够验证一个程序是否正确。尽管总体而言这肯定是无法实现的，但是关于这个问题还是进行了很多研究。至今为止的所有这类工作，在程序调试方面都还没有得到任何真正的实际应用。但是对于程序调试员来说，读一些这类文章将肯定会有一些作用。还没有关于这方面工作的权威性总结，也不可能写得出来，因此这份参考资料已经足够了。

□ Weinberg, G. M., and G. L. Gresset, An Experiment in Automatic Verification of Programs, *Communications of the ACM*, 6, 10 (October 1963), pp. 610－613.

在这项早期研究中，作者试图揭示编译器的错误检测属性，如何取决于它们被使用的错误环境；同时他们也针对实际环境中错误检测的性能问题，提出了一系列的建议。但是这些建议中，被主流编译器所采用的寥寥无几。

□ Irons, E. T., An Error Correcting Parse Algorithm, *Communications of the ACM*, 6, 11 (November 1963), pp. 669—673.

□ LaFrance, Jacques, Optimization of Error Recovery in Syntax - Directed Parsing Algorithms, *ACM SIGPLAN Notices*, 5, No. 12 (December 1970), pp. 2—17.

关于使用编译器自动更正语法错误的问题，Irons 的文章是最早的，而 LaFrance 的文章则是最新的。这些“更正”的方案背后的思路都使用了所谓的“接近性”概念，也就是说一旦发现错误，就通过尝试性的构造来更正。这一领域的工作者可能会喜欢使用“接近性”概念，因为这个概念不会受用户心理的影响，但是这种希望似乎只是空中楼阁。至少在一开始的时候，只有首先通过对使用该语言和操作系统进行工作的程序员进行实际研究，然后把他们最经常犯的错误进行分类，最后再提供一些专门的程序来修正这类错误，我们才能获得较大的进展。比如说在 Cornell 大学开发的 PL/C 编译器，为了保证校正并且成功编译一个 PL/I 程序，主要的一个步骤实际上很简单，这就是对程序员进行一些限制，禁止他们跨卡片地书写注释或者字符串。这种方法可以限制住错误的传播范围，同时在进行更正或者至少是进行更有意义的编译等操作时，也极大地增加了操作成功的可能性，从而使得只需再运行一次，就可以更正所有的问题。只有在首先使用这些特殊的方法，消除了导致错误的主要心理原因之后，我们才有必要去采用这些文章所建议的更为精致的方法。

□ Teitelman, W., PILOT: A Step Toward Man-Computer Symbiosis (Doctoral Dissertation, M. I. T.), Cambridge, Mass., June 1966.

PILOT 系统是用 LISP 语言写成的一个工具，其目的是为那些负责修改 LISP 程序工作的程序员提供协助。虽然该系统有很多非常有趣的特点，但是我们还不清楚，对于那些习惯于使用该系统的程序员来说，该系统会有多少帮助；我们也不清楚，与简单的错误检测手段相比，其帮助作用有多大。这方面的继续研究将很有用处。

□ Stockham, Thomas G., Jr., Some Methods of Graphical Debugging, *Proceedings of the IBM Scientific Computing Symposium on Man-Machine Communication*, 1966.

Stockham 回顾了使用 CRT 显示设备进行动态调试的方法，并且至少追溯到了 1958 年。面对这些早期实验的结果没有能够达到实用水平的事实，他发出阵阵哀叹，并且将失败归咎于“一系列的技术以及经济原因”。然而实际情况可能是，因为缺乏确凿的关于这些系统所能带来的益处的行为数据，所以也缺乏促使人们去克服技术和经济方面的这些困难的动力。毫无疑问地，在低效率的程序调试中蕴藏了足够的经济潜力，只要我们能够使人们相信在这方面可以获得好处，就会使这种潜力成为技术发展的动力。

□ Mills, Harlan D., Syntax-Directed Documentation for PL360, *Communications of the ACM*, 13, 4 (April 1970).



Mills 介绍了一个更加雄心勃勃的计划，根据该计划，将把文档所需的素材当成与计算机交互过程中的一部分，在编写程序的过程中抽取出来，然后在稍后一些时候，通过计算机的处理得到文档。这种方法似乎在两个方面触及到了问题的本质：在生成文档的过程中，这种方法似乎并不要求人们有很大的积极性，同时也不需要打印出一大堆文档——而以往的人们既要费力去检查这些纸面的文档，而且没有任何工具可以借鉴。尽管如此，从某种意义上说，这确实是一般的计算机用户解决这类问题的方法——“让我们来看看，我们是否可以让计算机自己来做这件事。”这种想法应当值得去尝试，不仅要看看它是否真的会生成可供使用的文档，而且也要看看这些是否确实是我们所需要的文档。

□ Klerer, M., and J. May, *Reference Manual*, Revised ed., Dobbs Ferry, N. Y., Columbia University Hudson Labs, July 1965.

这份文件的篇幅只有两页，希望了解 Klerer-May 系统的用户可以从这份文件开始学习。整个文档外加 1 页附录，后来重新出现在 Sammet 的“Programming Languages”一书之中（参见前一章末尾的参考资料）。

□ Leeds, H. D., and G. M. Weinberg, *Computer Programming Fundamentals/Based on the IBM System/360*, New York, McGraw-Hill, 1970.

除了程序开发的书籍中泛泛的讨论之外，在本书中你将见到更为广泛的关于程序文档工作的讨论。其中的一章论述了流程图问题，另一章则讨论了书面记录问题，通过与相关问题的联系，文档工作的其他方面也有充分的分析。然而，这些成果还远非关于这个问题的最后的完整结论。

第13章　其他的程序开发工具

评　　注

如果是在今天来写这本书，我将会把关于程序语言的那两章删减为两段，而同时将把有关工具的那章扩充为5（甚至是10）卷。在关于第4篇的评注的一开始，我曾经过分夸大了当前的工具（过分夸大的工具?），当时我所指的并不是程序语言，而是很多的其他的工具——在1/4个世纪之前，我们曾经下意识地把它们看作是次要的工具。

与其他工具相比，人们对程序语言具有带着成见的偏爱；这种现象长期存在，一直延续至今——只要我们考虑到测试以及测试工具的问题，理解这个问题简直是再容易不过了。在初稿中充斥着我们的梦想，而这些梦想都是关于当时正在制造的以及我们渴望制造的工具——代码覆盖分析器、静态和动态结构分析器、测试数据库、规范说明语言、测试装具模块以及驱动程序——这些东西在今天都成为了现实，并且数量繁多，甚至包括那些我们在过去想都不敢想的东西。如果我们不使用这些工具，那么质量低劣的代码中就存在着总体风险；然而就这个意义而言，这些工具在实际开发工作中得以应用的程度是微乎其微的。至今依然保留下来的，除了完全利己程序开发方式问题之外，还有专业测试人员的社会地位问题。显然地，通过改良测试工具，并不能找到解决这些问题的办法；真正的解决之道应该从改进管理入手。

个人电脑的出现，使得程序语言与操作环境之间的界限变得模糊了。这样，原书中本章的这两节中的许多要点似乎变成了关于方便程度的问题，而与基本心理方面的重要性无关。对于那些执行起来不方便的

工作，人们依然会更加乐于暂时把它们放在一边。为了实现软件任务的集成化，我们付出了许多努力；虽然如此，这些集成工作中的相当部分依然是由人自己来完成的，这时的人所扮演的角色就是他们使用的计算机操作系统——比如说他们也许会使用脚本描述语言，这时他们自己就转换成为元操作系统的设计者。

我们需要为软件开发人员提供完全集成化的开发/维护环境，无论我在 1971 年写了些什么，在这一方面我们还任重道远。如果说我们可以透过本书的前后两个版本得到一些指导的话，那就是，也许我们甚至还看不出这种集成化存在的任何可能。在集成化向前推进的过程中，我们不可能预测出将会出现哪些行为；我们可能预测的只有一点：我们在今天认为是极为重要的程序开发工作，有朝一日会变成无关轻重的家务管理性工作。如何证明这种预测呢？程序开发在过去的历史，就说明了这种由重要向非重要转变的过程。

我们也可以预测出，在他们的基础性工作被重新定义之后，今天的大多数程序员会感到不快——只要设身处地地体会一下我们自己在这种情况下的感受，就会很容易理解这一点。用两根橡皮筋把程序卡片捆成两摞，然后托在双手上；接下来仔细地拆去橡皮筋，把成摞的卡片放入读卡器中；再按下按钮，就可以在一旁看着卡片在托盘中堆积起来；当最后一张卡片也堆好后，按下 end-of-file 按钮；接下来可以把所有卡片取出来；如果穿孔时遗留了一些孔屑，可以轻轻弹去；然后在特制的平台上摆放好；再换上新的橡皮筋；最后就可以把整齐的卡片组放回到自己的桌子上了——嗨！在这整个过程中，我们的心中的确会有某种满足感。这是真实的集成化，而我们中有些人会怀念这种施予控制的感觉。

嗯，实际上我却并不怎么怀念卡片（而对于纸带，我的怀念就更少了!），虽然如此，我却渴望获得某些不得已延迟——虽然有人对这类渴望颇有微词，但是这样的批评我实在是听腻了。那些不得已的延迟，给了我时间坐下来思考自己正在做的工作。无论未来的程序开发会如何发展，我还是坚信独立思考要比完全依赖计算机、懒得动脑子收获更大。我所持的这种观点，是否会让别人认为我是一个不可救药的老顽固？换

言之，为了帮助那些希望有时间进行反省的程序员，我们是否应该多做一些事情呢？而为了帮助那些不愿意反省的程序员，要做的事情是不是更多呢？

对于我们中的大多数人来说，分时计算已经成为一个垂死的问题，技术的发展“仁慈地”杀死了这个问题。在当时它为我们做的就是提出了界面设计的问题，也就是 Tom Gilb 和我在“*Humanized Input*”一书中所探讨过的问题。在当时这是一本开创性的书，当然，随着技术的发展它很快就变得过时了。虽然如此，界面设计当然还是一个有待进一步研究的问题——而且在我写作本书的最初版本时，这也是我所感兴趣的心理学研究的一个部分。今天，我们已经有了针对人机交互的学术杂志和专业研究团体，尽管它们研究的，都还不是那些在界面背后的程序开发或者程序开发的其他心理因素。

我个人认为，对于程序员的工作，界面方面的研究结果似乎没有产生多少直接影响。首先，工具的设计者以及过程的设计者似乎都是基于直觉，而不是科学的结果进行工作的。其次，这项研究的结果显示，人的适应性非常强，他们可以使用几乎任何形式的界面进行工作——并且会认为这些界面都是对用户友好的。最后，即使说这些直观设计的界面之间有何差别的话，这种差别也很小，不至于在程序开发这样复杂的工作中成为一个主要的因素。

下面一个问题就是文档。本书提供了某种形式的文档，在经过 25 年之后，我已经把首次出版时我所遇到的大部分问题忘掉了。我也的确记得本书开始流行之后的那段美好时光；但是即使这本书没有走红，我也会为曾经写出了它而高兴。不用说文档所具有的其他价值，光是令你能够回忆起自己做过的事情，从而更好地对自己下一步的工作做出预测，就已经是很美妙的了。

就软件的文档而言，在过去的 25 年间我们似乎并没有取得多少进展。不错，我们确实拥有了一些灵巧的新工具（比如代码格式化程序、在线文档以及超文本等），但是，最重要的还是**内容**，然而很少有工具会提供真正的增强了的内容。就像在 Internet 中经常出现的情况一样，

在首页上罗列一大堆易于访问的链接，指向一些你根本就不想要的信息，这种做法没有什么用处。

只有通过制度化，使文档的制作过程转变为软件行业中的一种专门化的工作，才会对文档内容产生最大的影响。有许多的商用软件开发单位，就是不要求开发者去写文档——当然，不是用户文档。因为他们认为，建立文档的工作不应该由软件开发者来承担。这正如我在当时所写道的：

> “……在讨论计算机程序文档的问题时，我们根本就没有涉及到计算机程序开发心理学——只有程序员们的一种误解是个例外，他们总是错误地认为，虽然不一定每个人都能够成为程序员，但是任何人都可以毫不费力地胜任制作文档的工作。”（p. 367）

Chinker

第V篇

程序开发心理学

Computer Programming Psychology

结　语

如果一直读到了本书的末尾，那么你就有资格得到作者的赞扬，因为你给予了一位作者能从读者那里所获得的最大荣幸。不过也许和那些半途而废的读者相比，这些读者并不希望从作者那里获得多少赞誉之词——因为，倘若能够把一本书完整地读下来，那么你就肯定是被它所打动了，而任何口头的赞扬都不可能与这种切实的收获相提并论。曾经有一位评论家这样说过："我们可能整个下午都坐着阅读一本书，在阅读的过程中我们会感觉很愉快，但是当我们起身离座时，也许会觉得'浪费'了时间——这正如那个古老的笑话所说的：'只有在把头转过来之后，你才会意识到剃刀有多快。'"尽管如此，或许在读完本书之后，并不见得每个人都会受到感动。当然，如果这位评论家在比喻中所用的言辞不是这样刻薄，那么我肯定会感到更加好受一些；尽管如此，我还是不得不承认，这个评论的确道破了本书的写作意图。我的学生们也有相同的反应。如果参加过某次这样的讨论会，那么在一年之后学生们的评价通常是这样的："在谈论这些问题时我们会非常愉快，但是等我回到自己的办公室，哎唷！那里却完全是另一码事。"

学生们学会的最重要的东西，并不是行为科学方面的任何具体结论，也不是可供在考试中鹦鹉学舌式的照搬照套的某某定律的名字。他们真正学会的应该是，在自己的整个生命过程中，都会有一台配备精良的机器相伴在他的左右，可以用来对行为进行观察分析——然而遗憾的是，这台机器却没有派上用场。在一个计算中心里有这样一个不知名者留下的话这样说道：

头脑的能力是很强的，而我们只用到其中的10%——其原因在于我们可以使用操作系统。

因此，读者在读过本书之后，应该不只是会开始关注计算机操作系统的问题；要是果真被本书所打动，那么他们就应该开始研究围绕自己的中央处理器——头脑——的操作系统。果真如此，他们也就得到了最大的收获。

但是，如果我们使自己的操作系统变得更加高效，如果我们更为细致地观察我们自己以及我们周围的事物，而且如果这种高效率和观察收获使我们能够成为更加多产的程序员，这又会有什么用呢？"三个愿望"的寓

言从多种文化和古老的历史中给我们一个启示："要是自己最大的愿望果真实现了，接下来你又该如何呢?"要是可以通过心理学的、社会学的和人类学的研究，或者是通过简单的认识提高，我们果真能够成为更加出色的程序员，那么我们又应该到哪里去为自己的智慧找到用武之地呢？要知道，如果某件事根本就不值得去做，那么无论做得多么出色都肯定是没有意义的。

那么，我们正在使用计算机进行的工作是否值得呢？你正在使用计算机进行的工作是否值得呢？计算机是如此令人着迷的一个怪物，而程序开发又是这样一场游戏、一个玩笑。正因为如此，我们这些使用计算机进行程序开发的人就很危险——因为在不知内情的情况下，我们可能会成为某些人游戏中的一个小卒子，而他们所玩的游戏的结局将会让人笑不起来。如果当年希特勒拥有计算机，那么毫无疑问地，他将首先把这种工具用于更密切地监视犹太人和吉卜赛人，以确保他想送进焚尸炉的那些人将无一漏网。倘若当年的 Pilate[①] 拥有一台计算机，那么毫无疑问地，他将用计算机来收集告密者的信息；如果有人由于异教的狂热而胆敢对酷刑有所微辞，那么这种使用计算机的方法要比直接把他们钉死在十字架上更为有效。即使是在今天，就在我们国家的某些地方，有些人虽然正在表面上和别人一样地使用着计算机，但是在他们的兵工厂内却因此可以不断地制造出更强大的武器，凭借这些他们可以迫使其他人服从自己的意愿——服从他们所谓的一个人应该如何正常生活的观念。

实际上，当今的情况与 1939 年或者基督纪元的一开始时没有什么区别。总有人由于被乐趣与利益所驱使，甘于出卖自己的劳力与脑力，从而为另一些人所利用，助纣为虐——在看过了上面那么多例子之后，难道你还会怀疑这一点吗？可能捧着这本书的，正是某双甘愿为人利用的手；而他们之所以阅读这本书的目的，正是为了其雇主的利益。

多年以前，也就是在我和 Herb Leeds 一起完成我的处女作之后的几年，我就曾经读过一篇关于某些实验的文章，这些实验用不同剂量的毒气来熏猴子，用来验证需要熏多久才能杀死它们。这些工作是在一个进行化

① Pilate，比拉多，"圣经"中钉死耶稣的古代罗马的犹太总督。——译者注

学战研究的实验室内进行的；其目的很显然，无非是根据猴子的数据来推断人。根据我的记忆，共有 36 只无辜的猴子以这种令人毛骨悚然的方式魂归西天；或许在将来的某一天，将会有3 600万个人以更为高效的方式命丧黄泉。在那之后的好几个月内，这篇文章一直萦绕在我的脑海中；实际上时至今日，这个阴影仍然在我的心头挥之不去。我估计是纯属巧合，在一次会议中有一位友好的年轻人和我搭话，他不惜血本地对我大加奉承了一番，并且极言他在读过我的书之后，得到了如何如何大的帮助，并且因之成为了一名更加出色的程序员。但是当我问他所从事的工作时，他却告诉我他在某个进行化学战研究的实验室工作。

虽然一想到这方面的研究我就反胃，但是这以后我还是提醒自己应该更加理性一些，所以我假想他所效力的那个实验室应该不是做过那种残忍实验的那个——但愿真的不是；或者他从来没有参与过这些对猴子的实验——也但愿他真的从来没有。但是我清楚地知道，在某些地方，某些人一边不断从我的书中获得知识，一边却正在参与这类实验，甚至实际情况比这还要糟。我意识到自己也承担着责任——这种责任告诉我：虽然写作一本书是一种教学的手段，但是如果我们对其后果不了解，或者难以想像其后果，那么写作就不仅仅是一种教学手段。在此后相当长的一段时间内，我一直没有精神去提笔写作，也许就是因为这个缘故，但也可能是由于其他原因。但是最后，我的本我终于战胜了自己，于是我又开始奋笔疾书了；这时的我已经决定要尽力保证只有那些纯洁的心灵，才能从出自我笔下的书中得到帮助。

在一定程度上，这本书已经实现了这个目标，因为对于某些类型的人，我并未指望能够唤醒他们，并使他们承认“程序员也是人”这个观点；在读这本书时，这部分人或者干脆半途而废，或者不能从中得到任何帮助。我过去曾经相信，心眼好的人不可能制造出差劲的系统。然而到现在我才意识到，我过去的这种想法过于天真——果真像我所想像的那样，为什么几乎我所遇到的所有人都那么好，但是同时却还会见到如此之多蹩脚的系统呢？不，肯定还需要有其他的什么东西，而且这种东西是一位作

者无法通过书籍传递给读者的。正如 Malraux[①] 曾经说过的："我们毕生的使命，就是要造就另一个人。"从这个角度来讲，写一本书只是这项使命中不足挂齿的一部分——而剩下的部分，应该由你来继续完成；而这项使命，将是永远也没有尽头的。

虽然我已经说了这么多，但绝不对不是要为自己在写作本书过程中的所作所为，找一个逃避个人责任的借口。我们正站在一个新时代的边缘，正是由于蕴藏于计算机之中、迟早要发生的革命，这个时代才有可能到来。站在这个边缘，我们有两个可以选择的方向——迈入自由的黄金时代，抑或是倒退回专制的黑暗年代——无论你如何选择，这个世界中任何已知的事物都将被人类征服。也许任何个人的努力对最终的结果都不会有什么影响，但是我们绝对不应该放弃尝试，因为否则其后果必然是回到专制。这本书就是我反抗专制、反抗人奴役人、反抗被自己的无知所奴役的一次努力。但愿专制的力量不会利用这本书，但是毫无疑问，专制的力量确实会借鉴这本书。既然在这方面的希望破灭了，那么我只能希冀，处于杠杆另一端的正义力量能够从本书中获得更大的帮助。

① Malraux，马尔罗·安德鲁（1901—1976），法国作家和政治家。二战时法国抵抗组织的成员，1959—1969 年出任查尔斯·戴高乐（Charles de Gaulle）政府的文化部长。他的作品包括历险小说和政治活动小说，例如"人类的命运"（Man's Fate，1933）以及一些有关艺术史的书。——译者注

第V篇 结 语

评 注

这篇结语经受住了时间的考验，我不需要增加任何东西。

然而，作为这次25周年纪念版中新的结语，我愿意邀请你来做两件事，它们是我在25年之前没有能够完成的两件事。

首先，如果你希望及时了解我正在做什么、正在思考什么、正在写什么以及正在教什么，我愿意邀请你访问我的主页：

http：//www. geraldmweinberg. com

其次，如果你愿意与我分享自己的思想、观察结果或者经验，也欢迎来到我的主页，并且给我发送电子邮件。只要我还在这个世界上，你就可以随时通过这种方式与我联系。我对自己很有信心，相信能够在和大家告别之前有足够的时间来完成“*Psychology of Computer Programming*”的金年纪念版。

策划人语

曾几何时，计算机在人们心中是多么的神圣！人们坚信，通过科学家们的卓越努力，我们身边的一切问题都可以交给计算机来完成。

程序员温伯格认为，计算机程序开发是一项人类社会活动。

曾几何时，计算机程序员在人们心中是一个多么神圣的职业！人们坚信，只有极具天赋的天才才能够编写出完美的程序，也只有天才才能够驾驭神秘的计算机。

系统设计师温伯格认为，优秀的程序员是培养出来的，而不是天生的。

我们常常能够感受到周围世界与我们最初的设想不尽相同，我们将寻找冲突根源的镜子指向别人，试图从指责的快感中找到责任的真正承担者。

咨询师温伯格用他精辟的论断提醒你：当你用一个指头指向对方的时候，留心一下其他三个指头指向谁。

我们常常为各种各样的问题所困扰，有时候，我们不知道自己在干些什么、应该干些什么，甚至不知道自己真正想干些什么。

思想家温伯格用他精辟的论断提醒你：问题其实就是你的期望和你的感受之间的差别，而且不管看上去如何，人们很少知道他们想要什么……直到……你给了他们所需要的东西。

成熟的软件产业是管理实践和信息技术结合的产业。而软件管理思想的引进，比技术和资金更为紧迫和重要。在选择国外优秀的软件管理图书的过程中，我们不断地接触到杰拉尔德·温伯格（Gerald M. Weinberg）的大名，温伯格的思想被难以计数的专业文献重复引用，众多的机构和专家强力推荐他的著作，他的每本著作都非常畅销。因此，温伯格先生和他的书引发了我们强烈的兴趣。

温伯格首要的贡献集中于软件领域，他是个体心理、组织行为和企业文化角度研究软件管理和软件工程的权威和代表人物。在超过 40 年的软件职业生涯中，温伯格从事过软件开发，软件项目管理、软件管理教学和咨询，他更是一位杰出的软件专业作者和思想家。

1997 年，温伯格因其在软件领域的杰出贡献，被美国计算机博物馆的计算机名人堂选为首批 5 位成员之一。这个名人堂至今只有 20 名成员。为中国读者所熟悉的比尔·盖茨和迈克尔·戴尔也是在温伯格之后方才获得这一计算机界至高无上的殊荣。

温伯格精力旺盛、思想活跃，从 20 世纪 70 年代开始，他总共撰写了 30 多本书籍和数以百计的论文。这些著作主要集中在两个主题：人与技术的结合；人的思维模式、思维习惯以及解决问题的方法。在西方国家，温伯格拥有大量忠实的读者群，这些“追星族”阅读了温伯格的每本重要著作，他们甚至建设有专门的组织和网站，讨论和交流大师的重要思想。可以说，温伯格近年来的每本新书都是在万众瞩目中推出的。

在经历了漫长的选题论证、版权联络、组织翻译和编辑加工过程，我们终于出版了这位大师最重要的 14 本著作。我们真诚希望这 14 本著作能让读者有所收获。感谢你们的阅读。

清华大学出版社·闻洁编辑室

2003 年 5 月